The Dissertation Committee for Maysam Lavasani
certifies that this is the approved version of the following dissertation:

Generating Irregular Data-stream Accelerators: Methodology and Applications

Committee:

Derek Chiou, Supervisor

Jacob Abraham

Eric Chung

Andreas Gerstlauer

Keshav Pingali

Generating Irregular Data-stream Accelerators: Methodology and Applications

by

Maysam Lavasani, B.E.; M.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2015

Dedicated to my family...

Acknowledgments

I am thankful to many people who helped me during the years of pursuing my PhD. I could not have finished my thesis without their help and support. First, I like to thank my advisor, Prof. Derek Chiou. Derek gave me a very interesting problem to work on and was open to the specific angles of the problem that I liked to focus on. He taught me the fundamentals of presenting my research in an effective way. The combination of his helpful and positive personality and technical interest made him the best advisor I could have.

I also like to thank my parents, Monir and Sadri. They had to work hard during the early years of their marriage to raise me. Also, they were always extremely helpful and encouraging during my long years of studying to follow my dreams.

During our years as graduate students, my wife, Shahrzad, and I were working hard on our research. We were also raising our son, Shahriyar. Shahrzad was parenting more than I, for which I am very thankful to her. Shahriyar was very helpful to me because he was always inspiring us in hard times. Also, my brothers and sisters, Sareh, Setareh, Mehdi, and Alireza were always supportive.

While I was searching for other application domains that could benefit from my research, Dr. Eric Chung and Dr. John Davis introduced me to interesting big-data applications. At the same time, Prof. Jonathan Bachrach introduced me to Chisel, an open source project that was essential to achieve

my goals in my research. Also, Prof. Andreas Gerstlauer and Dr. Behnam Robatmili gave me important pointers to related work. I like to thank them all as well as the members of my dissertation committee for their valuable feedback.

I have spent a lot of time talking to Hari Angepat, my group-mate, about my research. I am thankful to him because of his insightful feedback. Also, thanks to my other group-mates, Yi Yuan, Gene Wu, Xiaoyu Ma, Luis Pena, Dan Zhang, and Lauren Gauckert for their help.

Maysam Lavasani
May 2015, Austin, TX

Generating Irregular Data-stream Accelerators: Methodology and Applications

Maysam Lavasani, Ph.D.
The University of Texas at Austin, 2015

Supervisor: Derek Chiou

This thesis presents Gorilla++, a language and a compiler for generating customized hardware accelerators that process input streams of data. Gorilla++ uses a hierarchical programming model with sequential engines run in parallel and communicate through FIFO interfaces. It also incorporates offload and lock constructs in the language to support safe accesses to global resources.

Beside conventional compiler optimizations for regular streaming, the programming model opens up new optimization opportunities including (i) multi-threading to share computation resources by different execution contexts inside an engine, (ii) offload-sharing to share resources between different engines, and (iii) pipe-offloading to pipeline part of a computation that is not efficiently pipelinable as a whole.

Due to the dynamic nature of Gorilla++ target applications, closed-form formulations are not sufficient for exploring the design space of accelerators. Instead, the design space is explored iteratively using a rule-based refinement process. In each iteration, the rules capture inefficiencies in the design, either bottlenecks or under-utilized resources, and change the design to eliminate the inefficiencies.

Gorilla++ is evaluated by generating a set of FPGA-based networking and big-data accelerators. The experimental results demonstrate (i) the expressiveness and generality of Gorilla++ language, (ii) the effectiveness of Gorilla++ compiler optimizations, and (iii) the improvement in the design space exploration (DSE) using rule-based refinement process.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

At the time of writing this thesis, one of the major challenges in information and communication industry is dealing with the large amount of data, generated from social applications, sensor applications, business automation systems, interactive multi-media systems, and security applications. This is known as **big-data** phenomenon. According to IBM [39], 1.8 zeta bytes of data were generated in 2011, and is doubling every year. Such a large amount of data requires higher performance data center infrastructure from networking, to storage, to computation. Consequently, big-data applications are one of the major growth drivers of the server class, microprocessor market.

In a general-purpose microprocessor, the overhead of instruction processing is much higher than the actual operations performed by each instruction. This overhead includes the necessary steps to fetch and decode the instructions, provide required operands for the instructions, and perform the necessary bookkeeping to ensure correctness when multiple instructions are executing in the microprocessor. Application-specific hardware, also known as accelerators, are faster and lower in power consumption than general-purpose processors because they eliminate most of the overhead of a general purpose processor [17, 28, 33]. As the result, using accelerators for emerging applications is an active area of research. Although fixed-function accelerators are more energy efficient than software running a general-purpose processor, they

are not a suitable solution for applications that change frequently.

FPGA-based accelerators [16, 46, 50, 51, 70], however, can be used to provide both performance and flexibility for such applications by trading part of the performance gain for the reconfigurability of the implementation substrate. Although the flexibility of FPGAs enables changing the application by reconfiguring the accelerator, their programmability is still a major obstacle for their wide-spread use.

High-level synthesis (HLS) techniques [19, 25] have been proposed to improve the productivity of hardware designers by automatically generating the hardware from a high-level description of an application. An HLS tool, ideally, (i) captures various types of parallelism in the input application, (ii) builds the design space of various micro-architectures that exploit the parallelism, and (iii) explores the design space to find the micro-architecture with the lowest silicon area and/or power consumption.

Pure C-to-gates HLS techniques rely on capturing parallelism between fine-grain operations of sequential code by constructing the control and dataflow graph (CDFG) of the computation kernels. They use scheduling algorithms [44, 53, 68] to extract parallelism between the operations that are provably independent in the CDFG. Conventional HLS techniques for capturing coarse-grain parallelism, however, are less effective. Since the highest performance hardware exploits both fine-grain and coarse-grain parallelism, for many applications, the Quality of Results (QoR) of these HLS tools are often lower than that of manual design process using low-level hardware-description languages (HDL).

To extract coarse-grain parallelism, main-stream HLS tools accept parallel programming constructs and/or annotations. For example, CatapultC [13]

and Vivado [88] accept SystemC [67] processes and modules. Vivado, also, accepts parallel functions/code-blocks communicating through dataflow channels. Altera openCL compiler [1] accepts single instruction multiple thread (SIMT) programming model. Impulse-C [40] and Handle-C [34] accept a subset of communicating sequential processes (CSP) constructs.

For many applications, there is a large micro-architectural space that can exploit coarse-grain parallelism. Choosing an efficient micro-architecture that meets the target performance, however, is often both non-trivial and critical to QoR. One solution is to rely on the designer to specify the micro-architecture alternatives of the sub-components in the design in the form of design parameters. The design space is built as the Cartesian product of the parameter values across different sub-components. Then, a generic optimization algorithm, e.g., hill climbing [35] or simulated annealing [82], can be used to explore the design space. Figure 1.1-a illustrates this methodology[1] for generating an efficient micro-architecture. Generic optimization algorithms are used for exploring the design space of various micro-architectures in HLS tools [58, 75, 76][2].

There are two drawbacks for this methodology. First, it relies on the design parameterization by the designer. Second, although exploring the design space using a generic optimization algorithm is much faster than an exhaus-

[1]**The difference between an HLS compiler and an HLS methodology:** An **HLS compiler** is a fully automated tool that accepts the high-level description of an application and translates it to a lower-level representation. An **HLS methodology**, on the other hand, may need a programmer assistance including parameterization or annotation in the code. As described in this section, current HLS tools require assistance from the users to exploit coarse-grain parallelism in an efficient manner. Therefore, they are referred to as methodology rather than a compiler.

[2]The micro-architectures in these tools are not necessarily exploiting coarse-grain parallelism.

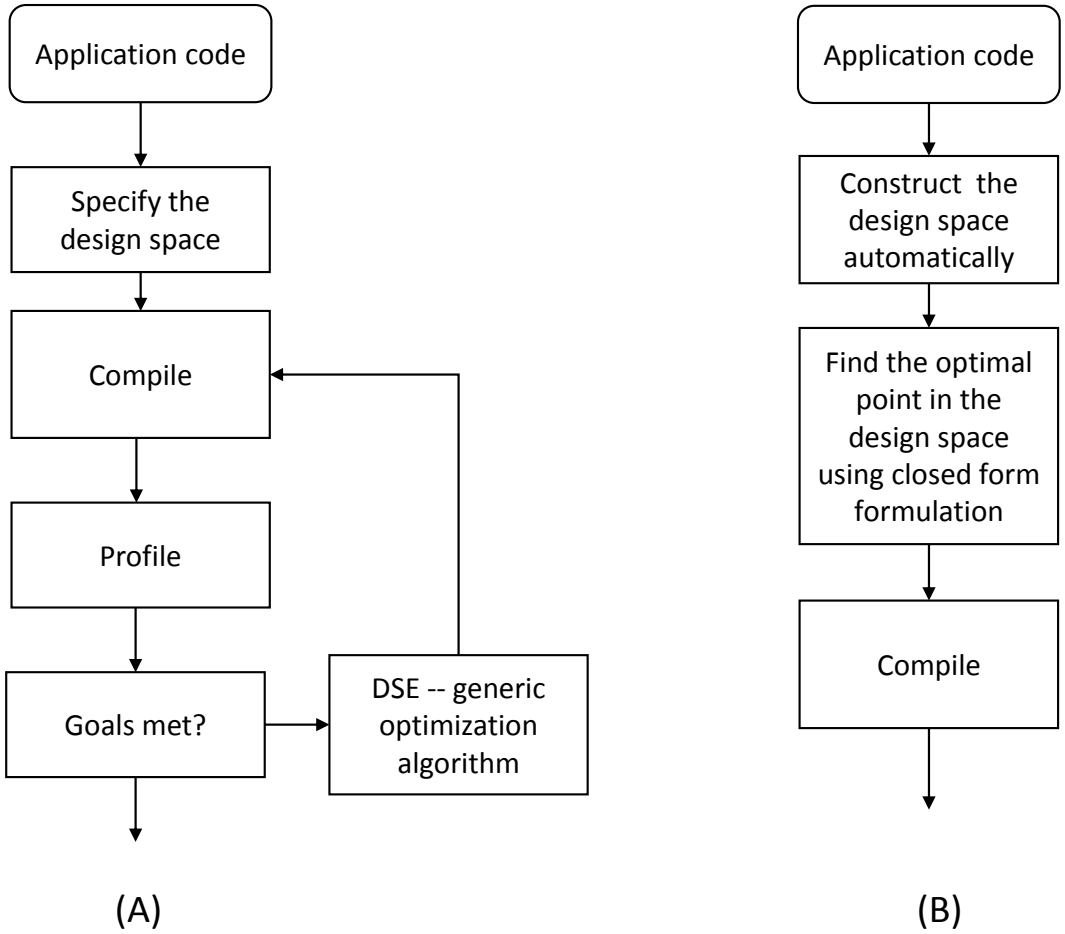


Figure 1.1: Two available methodologies for generating optimal parallel micro-architectures based on (A) an HLS with parameterized parallelization annotations/constructs + a generic optimization DSE and (B) an SDF-based HLS + an SDF-based DSE.

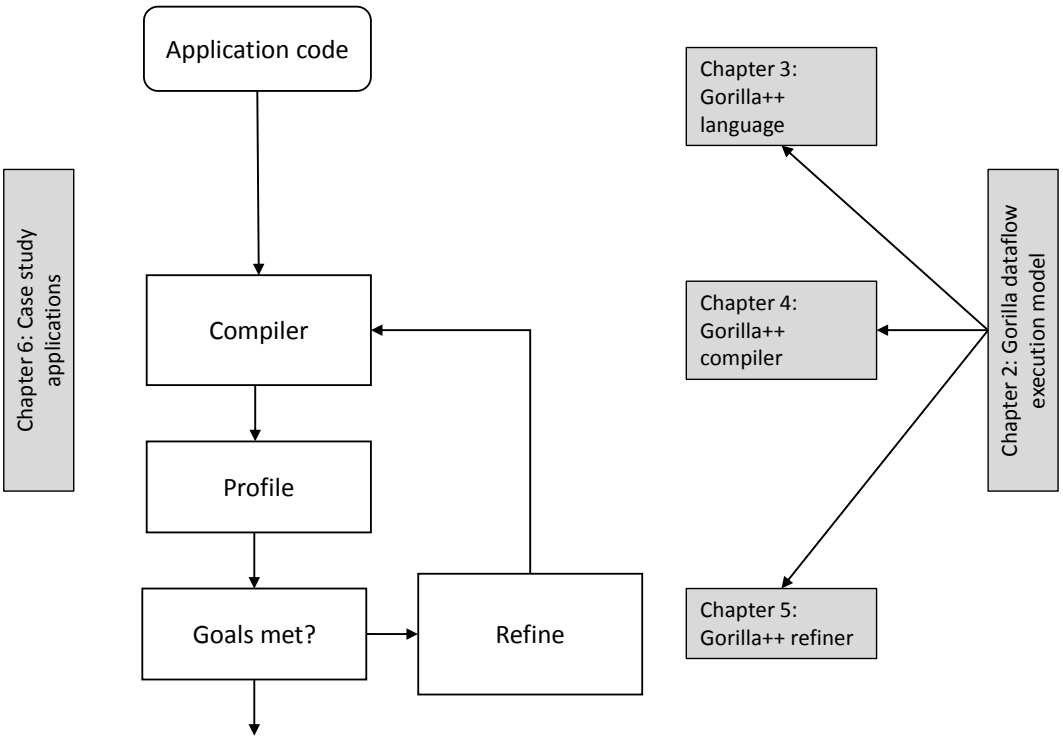


Figure 1.2: Abstract view of Gorilla++ methodology and outline of thesis chapters.

tive search, for many realistic designs, it still takes a long time. However, this methodology is attractive since it makes no particular assumption on the application execution model and consequently it supports arbitrary applications.

Unlike the main-stream HLS methodology, synchronous data flow (SDF)-based HLS [20, 30, 37, 54] uses a systematic solution to find the "right" micro-architecture to exploit the available coarse-grain parallelism.[3] They automatically build instances in the design space without requiring any user parameterization. These tools also have a model to predict the performance and the required silicon resources for a particular micro-architecture. Performance is predicted by balancing the production and consumption of tokens in SDF channels. Also, area is predicted based on aggregating the individual areas of the SDF nodes.

In this methodology, however, the number of input/output tokens and the execution time of each dataflow node for processing the input tokens need to be static and known at compiler time. Due to this static nature of the SDF model, an SDF-based HLS can explore the space of a design using closed-form formulations, resulting in micro-architectures that achieve throughput and area goals independent of the input data values. SDF, however, is restricted to regular applications such as signal processing and multi-media. Figure 1.1-b illustrates this HLS methodology which does not need any parameterization and also can generate the optimal micro-architecture using closed-form formulations.

[3]These tools are descendants of StreamIt [85] project which is a streaming language and compiler based on SDF execution model. It is designed with the purpose of implementing streaming applications on general-purpose multi-core hardware. However, later, researchers added a hardware generation back-end to the StreamIt compiler and used it as an HLS methodology.

Other streaming application spaces include networking and big-data where messages stream through different processing stages. They would also benefit from automatic refinements of micro-architectures. They, however, differ from SDF applications in two ways. First, the computation stages in these applications often require access(es) to global resources to process input tokens.[4] This is in contrast to the SDF model (and process networks [47] in general) where the input tokens are self-contained for processing. Second, since for each set of input tokens, the processing time and the number of generated output tokens are data-dependent, static formulations are not sufficient to predict the performance of the refined micro-architectures.

Gorilla++ extends the application of automatic micro-architecture refinements by eliminating the above two restrictions. It supports efficient and safe accesses to global resources using multi-threading and lock-based synchronization. It supports irregularity by replacing the static, formulation-based DSE with an iterative, refinement-based DSE. In each iteration, Gorilla++ (i) profiles the accelerator using a given input data-set, (ii) automatically detects bottlenecks or resource under-utilized resources in the accelerator micro-architecture, and (iii) eliminates such inefficiencies using generic refinement rules. Consequently, Gorilla++ can optimze the accelerator micro-architecture given a particular input data-set.[5] The generated micro-architecture might not be optimzed for other input data-sets however. Since the methodology is based on profiling hardware performance counters, it can optimize such micro-architectures even if part of the design is black boxed and/or does not have a simulation model.

[4] **Token** and **data element** are used interchangeably across this thesis.

[5] Like other heuristic-based optimization algorithms, the generated micro-architecure might not be optimal.

Figure 1.2 illustrates the Gorilla++ methodology. The Figure also shows the relationship between different parts of this thesis with different components in the methodology.

1.1 Problem statement

This thesis addresses **inefficiency of current HLS methodologies for building and exploring the design space of parallel micro-architecture alternatives in irregular data-stream applications.**[6] Due to this inefficiency, the current methodologies need to compromise either on productivity, generality, or quality of results (QoR).

1.2 Solution statement

The solution that this thesis presents for the stated problem is **an execution-model, a language, and a toolset with an intrinsic support for building and exploring the design alternatives using an iterative refinement process**.

Gorilla++ dataflow (GDF)[7] is an extension of process networks with

[6] **Clarification on the dataflow, streaming, and data-stream terms as categories of applications:** Both **dataflow applications** and **streaming applications** have similar characteristics to the target applications of this thesis. However, since these two terms are highly overloaded in different domains, in order to avoid any confusion, the term **data-stream applications** is defined in Section 2.2.8.

[7] **Clarification on the dataflow term:** The term **dataflow** has different meanings in different contexts. These different meanings, however, convey the same fundamental concept, a graphical representation of computation. When used in the context of models of computation, it refers to a network of concurrent processes that communicate through dataflow channels. As a framework for programming a large cluster of computers, it is used to specify the computations on a large set of data. It is, also, used in the context of compiler techniques as an intermediate representation for compiler analysis and optimization algo-

the support for accessing shared resources and the centerpiece of Gorilla++ methodology. A GDF-based application consists of sequential nodes that communicate with each other through either one-way push interfaces or two-way request-reply interfaces. The Gorilla++ execution model is covered in Chapter 2.

The Gorilla++ language is designed based on this execution model. The language uses sequential semantics for programming dataflow nodes and functional constructs for composing nodes together in a structured manner. Chapter 3 describes the Gorilla++ language.

A compiler is developed to translate application code to a registers-transfer level (RTL) model that can be converted into hardware using existing tools. It can improve the area utilization of accelerators using multi-threading, Pipe-offloading, and Offload-sharing. Chapter 4 describes Gorilla++ compiler.

The design space of a Gorilla++ application is explored using a micro-architecture refinement tool that iteratively refines the design based on generic refinement rules. Chapter 5 describes the Gorilla++ DSE tool. Finally, Chapter 6 presents the case study applications that are implemented using Gorilla++.

1.2.1 Thesis statement

A programming model with software-style, streaming and shared-memory constructs can naturally describe an important class of hardware, irregular data-stream accelerators. Using auto-refinement of the micro-architectures,

---

rithms. And finally, it is used in computer architecture context as a hardware architecture that can understand and execute a dataflow graph as a machine language. Throughout this thesis the **dataflow** term is mainly used in the first context as an execution model.

a tool-set can rapidly generate a high-quality hardware from the application code written in this programming model.

1.2.2 Contributions

The contributions of this thesis are:

- A programming model for designing the target accelerators using software-style high-level construct. A design in this model includes sequential engines run in parallel and communicate through FIFO interfaces with the support for safe accesses to global resources using lock primitives.
- A set of compiler optimizations, including multi-threading, pipe-offloading, and offload-sharing.
- An iterative method for refinement of irregular micro-architectures using generic rules that find and eliminate design inefficiencies, either bottlenecks or under-utilized resources.

Chapter 2

Gorilla++ execution model

This chapter first describes the execution models that are related to Gorilla++ execution model. It then discusses the characteristics of NBD (networking and big-data) applications which are Gorilla++'s target applications and defines their generic forms as irregular data-stream applications. Finally, it defines Gorilla dataflow (GDF) as the execution model[1] of Gorilla++ applications.

2.1 Related work: dataflow-centric models of computation

This section explains dataflow-centric models of computations, including Kahn process network (KPN), SDF , and dynamic data flow (DDF), as they relate to Gorilla++ execution model.

2.1.1 KPN

A KPN consists of concurrent processes that communicate through unbounded FIFO channels. In a KPN, reads from the input FIFO channels are blocking and writes to the output FIFO channels are non-blocking. Also, in

[1] "Execution model" and "model of computation" are used interchangeably across this thesis.

KPN, that there is no source of indeterminism in the computation kernels associated with processes. Kahn showed that the communication patterns (the values and orders of tokens transferred on each channel) in a KPN are independent of the latency of computation kernels or the latency of FIFO channels [47]. Therefore, for a given input token sequences, the token sequences on all intermediate channels and output channels are deterministic.

2.1.2 Static dataflow

An SDF network is a special case of a KPN that consists of dataflow nodes[2] communicating through bounded FIFO channels. In addition to the restrictions inherited from KPN, the number of input/output tokens to fire (execute) an SDF node is known at compiler time. An iteration is defined as the series of firings that returns the network's FIFOs to the original number of tokens. In order to have bounded FIFO channels, the SDF channels must be rate-consistent, i.e., the input rate to the channel is equal to the output rate from the channel. Lee and Messerschmitt presented [52] a systematic method to schedule an SDF with rate-consistent channels. A schedule consists of the order and number of firings in each iteration.

The scheduling can be done at compile time using closed-form formulations. Also, the maximum size of the communication FIFOs in an SDF and the number of initial data elements to guarantee the liveness of the network can be determined through closed-form formulations. Thus, SDF is a highly analyzable execution model. SDF is often used to model regular applications, including signal-processing applications. Another important characteristic of the SDF model is that if the nodes are stateless and the execution time for each

[2]A dataflow node is a restricted version of a KPN process.

node is known in advance, the SDF can be refined automatically to achieve a target throughput [85]. Therefore, a compiler can replicate a minimum number of nodes to achieve a certain absolute throughput.

2.1.3 Dynamic dataflow

Like SDF, DDF consists of dataflow nodes that communicate through bounded FIFO channels. However, unlike SDF, the communication patterns are not static. Because of this dynamic nature, the scheduling of DDFs may not be possible at compile time. Both static and dynamic dataflow follow the notion of sequential atomic rules for firing.

2.2 Irregular data-stream applications

In order to design an effective execution model for Gorilla++, it is important to understand the characteristics of its target applications. This section covers the characteristics of NBD applications and then defines irregular data-stream applications accordingly.

2.2.1 Computation on external streams of data

The main functionality of packet-processing systems is receiving incoming packets from input network interfaces, processing the packets, and sending them to outgoing network interfaces. Likewise, server applications including big-data applications receive requests/data elements from the network, process them, and potentially send the results to the network. Although the benchmark applications in this thesis use network interfaces for sending/receiving the external streams of data, the streams can be from other sources too. For

example, they may be received from a direct connection to a processor, a DMA engine that streams data from the memory, or other I/O devices like a storage device. As we will see in Section 6.1 in Chapter 6, this characteristic is one of the design rationales of **in-line acceleration**.

2.2.2 Data-parallelism

Packet-processing systems and big-data applications are both data-parallel in nature. A high-end, multi-chassis router system with multiple network processors has more than half a million independent threads [18]. Similarly, in a server application, there is a large amount of data-parallelism between the processing of different requests/data elements.

2.2.3 Irregularity in control-flow

The processing of different data elements in NBD applications depends on their values, as well as the values of the global memory. For example, the way a packet is processed is highly dependent on the packet protocol. Similarly, the way a request or a data element is processed is highly dependent on the request type, the data element value, or the content of data structures in the server.

2.2.4 Accesses to global data

Unlike many signal-processing and multi-media systems with pure functional building blocks, NBD applications require accesses to global data, e.g., lookup tables and performance counters in networking applications and hash tables and graph data structures in server applications.

The global data are shared among the execution contexts associated

with different data elements. Therefore, concurrent threads need a mechanism to read from and write to the data structures in a safe manner, e.g., through the use of lock/unlock constructs to provide mutual exclusion.

The distributed big-data applications may also require accesses to the persistent storage sub-system, hard disks, or flash storage. For the big-data benchmark applications that are used in this thesis, however, it is assumed that each system keeps the data structures in the system memory. Extending this work to support applications with accesses to persistence storage can be done using similar methods.

2.2.5 Randomness in the global accesses

Accesses to the global data structures usually (i) are based on memory addresses generated from the input data element values and/or (ii) require pointer-chasing operations. Therefore, a large number of memory accesses in these systems are random memory accesses. This type of access receives a less-than-usual benefit from sophisticated memory sub-systems, including cache structures, pre-fetching hardware, smart memory controllers, and row-buffer locality in the DRAM [24, 59]. [3]

2.2.6 Weak ordering constraints

Most of the routing applications in layer-2 or layer-3 can process packets in an arbitrary order. However, the packet-processing systems preserve the incoming order of the packets by reordering them before sending them to the network. This is to reduce the complexity of handling out-of-order packets in

[3]Most of the locality in accessing the memory in these applications is associated with spatial locality between accesses to a given data element.

the end nodes. Note that there are flow-sensitive applications at the layer-4 and above that have more strict ordering requirements. Deep packet inspection, for example, needs the processing of packets in the same flow, a session between processes in the client system and the server system, to be done in sequential order. However, even in these applications, there is no limitation on processing packets from different flows in parallel. Similarly, most server applications follow the same weak ordering pattern. Requests from different clients can be handled in parallel. Depending on the server application, even different requests from the same client can sometimes be handled in parallel. In big-data applications, a mix of unordered and ordered operations is required. For example, both K-means and PageRank which are used as benchmark applications in Gorilla++ contains multiple macro phases that needs to be executed sequentially. In each phase however, processing the data elements can be done in parallel. More aggressive techniques to make the applications with ordering constraints amenable for parallel execution is presented in [69].

2.2.7 Small, frequently-used computation kernels

An NBD application must have small computation kernels in order to be implemented as specialized hardware. An application with a large computation kernel needs a lot of hardware resources, which is not always feasible. This is in line with the trend of using simple computation kernels for big-data applications in order to make these applications scalable for larger amounts of data.

2.2.8 Definition of irregular data-stream applications

"Irregular data-stream applications" are defined as applications with the following characteristics.

- The application processes incoming streams of data elements[4] received from an external source and generates outgoing streams of data elements sent to an external source.
- The application consists of several processing stages. Each processing stage executes a computation kernel on each incoming data element.
- The computation kernels in the application may potentially access global resources, including global memory.
- The computation kernels are data-parallel in nature, i.e., the incoming data elements can be processed independently. Any ordering requirement between processing of different data elements and/or mutual exclusion when accessing the shared resources should be explicitly stated by the programmer using synchronization mechanisms.
- The computation kernels in the application may potentially (i) consume/produce an input-dependent number of input/output data elements per execution and (ii) have an input-dependent execution-time for processing different data elements.

[4]The data elements might have different natures in different applications. In a networking application, a data element might be a layer-2 frame or a layer-3 [66] packet. In a server application a data element might be a request from a client or a piece of data as part of a large collection of data in a distributed big-data application.

It is important to note that although the motivating applications in this thesis are focused on networking and big-data applications, the developed methodology can be used for other applications with the above characteristics. For example, as will become clear later in Section 3.5 of Chapter 3, a non-blocking cache for exploiting the locality of memory operations can also be modeled as a hardware that processes an input stream of memory operations.

2.3 Gorilla dataflow

GDF, the execution model for Gorilla++ applications, plays an essential role in the Gorilla++ methodology. GDF is designed based on three major goals: (i) generality to cover a wide range of applications, (ii) expressiveness to facilitate the modeling of the target applications, and (iii) analyzability to leverage the auto-refinement of micro-architectures using the Gorilla++ compiler.

GDF uses a rendezvous mechanism for communication between dataflow nodes. Its rendezvous mechanism is implemented using FIFO interfaces, adopted from the theory of latency-insensitive designs [12]. In addition to push-only, one-way interfaces, a GDF has two-way request-reply interfaces, also known as offload interfaces. It is important to note that each node with offload interface can be split to multiple nodes (potentially with additional data flow channels) and each offload interface can be modeled as two one-way interfaces. This way each GDF is transformed into a dataflow graph without requiring any two-way offload interface. However, GDF uses offload interfaces as first-order construct in order to improve the expressiveness and analyzability of the model.

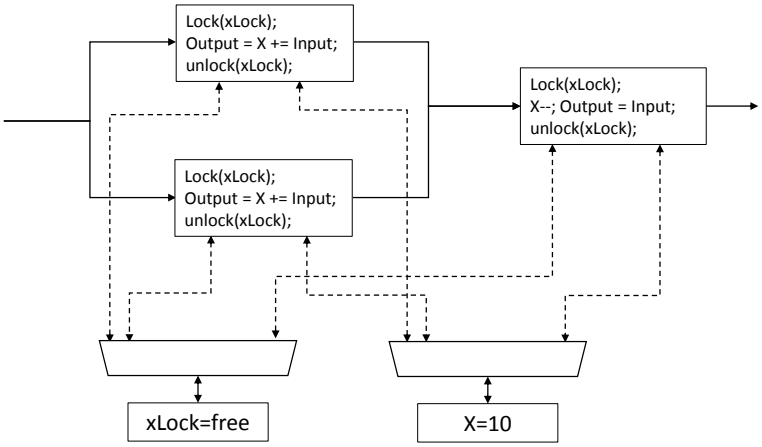


Figure 2.1: A GDF with global shared variable X. GDF nodes use lock to safely access shared offloaded data.

2.3.1 Handling shared resources and global state

In the KPN model, every dataflow node is supposed to receive self-contained token(s) that carry the data for processing. The node does not need any global states to process the incoming tokens. Gorilla++ target applications, however, need to access global states, e.g., shared data structures.[5] Gorilla++ uses a lock-based synchronization mechanism to solve this problem.

Gorilla++ uses single-copy, shared memory to save the global data. The dataflow nodes can access global memories through offload interfaces. Since multiple data-flow nodes access the shared memory, Gorilla++ requires that the programmer uses necessary synchronization mechanisms to ensure mutual exclusion while accessing the data.

[5]I-structures were proposed by Arvind and Pingalli [2] to provide a safe mechanism to access the global data in a tagged-token dataflow hardware. Gorilla++ is not following a tagged-token architecture.

Special lock engines are used for synchronization. A lock engine is, also, accessed through an offload interface. The lock engine does not reply to a lock request unless either (i) the lock is not taken in the first place or (ii) the lock is released and the requester of the lock is the winner among all other lock requesters. Section 3.4.1 elaborates on the details of the lock engine.

2.3.2 Structured dataflow composition

An important feature of GDF that improves both the programmability and the analyzability of the model is using structured composition. In a generic dataflow graph, a node can have any number of inputs/outputs with arbitrary connections to other nodes. In GDF, however, nodes' interfaces and connectivity are structured. A standard GDF node has one input, one output, and zero or more offload interfaces. Figure 2.2 shows a dataflow node with single input, single output, and some offload interfaces. Only special nodes with the sole purpose of merging or distributing data elements can have multiple input/output interfaces. These special nodes are transparent to the programmers, and are used when composition of nodes happens. They can only reorder the data elements; they cannot change the data elements themselves. Connecting the nodes is done using a predefined set of composition functions (see Section 3.4.2 in Chapter 3).

2.3.3 Multi-phase computation

There are cases in which a computation consists of multiple phases. In each phase, the dataflow nodes execute different computation kernels. A GDF programmer can attach the current phase to all data elements in the system in order to specify the changes in the computation phases.

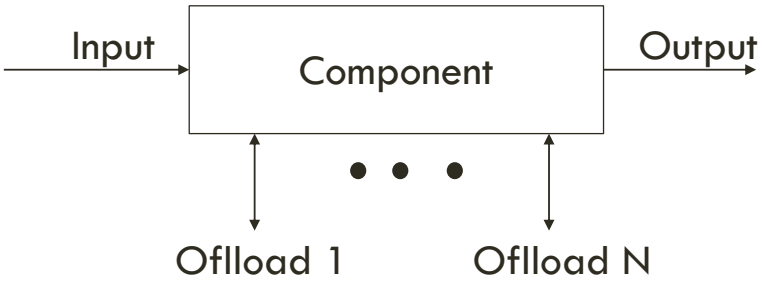


Figure 2.2: Each component has one input interface, one output interface, and zero or more offload interfaces.

2.3.4 Static allocation of dataflow nodes

Gorilla++ statically allocates [61] a GDF nodes in the hardware implementation. Therefore, for each node, a corresponding hardware component is generated and each channel is realized as dedicated buses between components. This property is not a characteristic of the GDF model; it is an specific choice for synthesizing a GDF application to a hardware. However, since static allocations affect the way a designer should reason about the possible deadlocks in the system we specify it in this section. It is possible to extend Gorilla++ and dynamically allocate and schedule the GDF description, i.e., time-share a hardware component to execute different GDF nodes.

2.3.5 Comparing GDF to other dataflow-based MoCs

The rendezvous-based mechanism in GDF can be transparently changed to a KPN communicating mechanism. As shown in Figure 2.3-b, for each channel between two GDF nodes, along the data, there is a forward "valid" signal, which indicates that the sender node has a data element to send. Also, there is a backward "ready" signal that indicates the receiver is ready to receive the

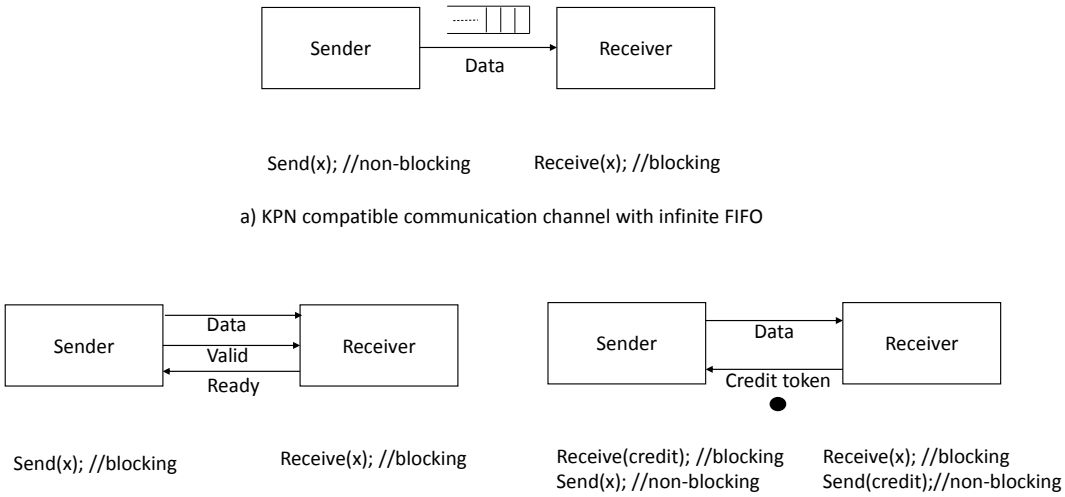


Figure 2.3: KPN communication channel, Gorilla++ rendezvous-based communication channel, and KPN-compatible communication channel equivalent to rendezvous-based communication channel.

data element. As shown in Figure 2.3-c, by changing the read and write semantics to a credit-based mechanism, the blocking characteristic of GDF only happens on the read operations – making GDF read/write semantics similar to KPN (blocking reads and non-blocking writes).

GDF, however, departs from KPN because it uses non-deterministic, eager, merger nodes for accessing shared global resources. These merger nodes are essential to guarantee deadlock-freedom when a lock engine is used in a design (Section 3.4.1 of Chapter 3). Unlike other dataflow-based models of computation and similar to StreamIt, GDF processes use structured communication rather than arbitrary communications (Section 3.4). Table 2.1 compares GDF and other related MoCs with respect to their communication and computation model.

Since a single computation node in GDF is a sequential state machine that (i) supports loops and conditional branches and (ii) can access memory, the computation node and the memory can simulate the behavior of a Turing machine [87]. The computation node simulates the Turing machine's state register, state transition rules, and head pointer and the memory simulates the storage for the Turing machines' tape. Therefore, assuming that we have infinite-sized memory, GDF is Turing-complete.

Since GDF does not fall into SDF model, refinement of the dataflow graph using closed-form formulation is not possible. Gorilla++ uses an alternative method for refining the GDF graphs which is described in Chapter 5.

Table 2.1: Comparing different dataflow-centric execution models in terms of communication mechanisms and communication patterns

Execution model	Communication mechanism	Communication pattern
Kahn process network (KPN)	Blocking reads and non-blocking writes through FIFO channels. FIFO channels are conceptually infinite in size.	Deterministic and dynamic
Synchronous dataflow (SDF)	Blocking reads and non-blocking writes through FIFO channels. Reads from channels follow atomic firing rules. Boundedness of FIFOs and deadlock can be decided statically.	Deterministic and static
Dynamic dataflow (DDF)	Blocking reads and non-blocking writes through FIFO channels. Reads from channels follow atomic firing rules.	Deterministic and dynamic
Gorilla++ dataflow (GDF)	Rendezvous mechanism between reader and writer (limited to structured communication)	Potentially non-deterministic and dynamic

Chapter 3

Gorilla++ language

Gorilla++ uses a hybrid language which consists of (i) a sequential, stylized C to specify the computation kernels associated with user-defined dataflow nodes, also known as engines, and (ii) composition constructs to specify the top-level connectivity between dataflow nodes.[1] This chapter covers the details of the Gorilla++ engine and top-level languages.

3.1 Related work: parallel programming constructs for high-level synthesis

This section describes the work related to Gorilla++ language constructs. We focus on high-level language constructs that enable an HLS tool to extract and exploit coarse-grain parallelism in the input application.

3.1.1 HDL-centric constructs

Ku and De Micheli proposed HardwareC [49], one of the first projects that extended the C language with parallel constructs to model hardware. It supports parallel processes that can communicate through message passing and/or shared ports.

[1] Lee and Parks followed Halbwachs [31] and called the composition language, the "coordination language". They also followed Jagannathan and Faustini [43] and called the node computation language the "host language".

SystemC [67] is a set of libraries that can be used to model a hardware design using C/C++. It contains the necessary constructs to define parallel hardware modules and processes. Although originally designed for co-simulation of hardware/software, the constructs are now supported in the synthesis process by a number of main-stream HLS tools [13, 88].

3.1.2 Bluespec: guarded atomic actions

Hoe and Arvind invented a rule-based hardware design language and compiler [36]. The rules are atomic operations predicated with a condition. They give the impression of freezing the rest of the system when a given rule's action is carried out after the rule's predicate is true. There is an implicit parallelism in this specification, because it is possible for multiple rules to be activated and executed in parallel. The compiler automatically schedules the rules such that they are either conflict-free or combined sequentially to preserve the promised atomicity semantic.

3.1.3 SIMT semantics

Due to the popularity of SIMT-style programming models to program GPGPUs, using this programming model for hardware design is an attractive solution [1]. SIMT assumes that a single kernel is executed for each piece of data. In order to tolerate memory latency (or even latency of execution pipelines), SIMT uses multi-threading.

3.1.4 Implicit parallelism in functional constructs

A large body of work [6, 21, 22, 55, 61, 80, 83] has pursued the path of generating hardware from functional descriptions of applications. There are

two major reasons that functional features can be beneficial in a language for high-level synthesis. First, side-effect-free operations enable safe parallelism among hardware components. Second, combinator functions can be used to implement regular connectivity patterns in the hardware. A subset of the research work which is closer to Gorilla++ functional composition constructs is covered in this section. An interested reader can refer to the survey paper [15] by G. Chen on functional HDLs.

Lava [7], invented by Bjesse, Claessen, and Sheeran, is a functional HDL, built on top of Haskell. It uses combinator functions to compose hardware components together. The idea of using combinators to express the connectivity between hardware components in Lava is inherited from previous relation-based HDLs, uFP [81] and Ruby [45], which were also invented by Sheeran and her collaborators. Lava uses recursive functions to describe regular hardware structures, e.g., a sort network or an FFT filter.

SAFL [61], which stands for "statically allocated functional language", is another Haskell-based language invented by Sharp and Mycsoft. It, statically, allocates a dedicated hardware component for every function definition. SAFL uses the function calls and, also, producer/consumer relations between the functions to model the connectivity between hardware components, i.e., the components corresponding to the caller and callee functions as well as the producer and consumer functions.

When a function calls another function, the caller is synthesized as a component that encloses the callee component. SAFL provides primitive hardware components which can be used at the leaf level of the design hierarchy. A sequential logic with feedback is defined using recursive functions. When there is a producer/consumer relation between two functions, i.e., the

result of a function is used as the argument to the other one, the output of the producer component is connected to the input of the consumer through a register. Later, the same authors introduced SAFL+, which borrows the notion of channels from CSP (see the next section). They also added arrays to support statefull components.

In spite of the advantages in functional languages, C-based programming languages (and their variations) are the main-stream choices for HLS tools. This is because, for many developers, the complexity of working with a purely functional HDL makes the functional advantages less appealing.

3.1.5 CSP constructs

CSP includes a language to represent sequential processes that run concurrently and communicate through rendezvous channels. It also includes a process algebra for reasoning about the concurrency characteristics of a CSP application, e.g., deadlock-freedom. Handle-C [34] and Impulse-C [40] are two HLS tools that add major CSP language constructs to standard C. SAFL+ also adds CSP constructs to SAFL to enrich its expressiveness.

3.1.6 Streaming constructs

Streams-C [27] consists of a set of C library elements that facilitate working with streams as first order objects. The compiler generates hardware from Streams-C description.

A series of HLS research projects [20, 30, 37, 54] followed the Stream-It project and adopted its SDF-based streaming programming model. Automatic refinement of micro-architectures in these projects is restricted to regular streaming applications.

3.1.7 Extending modern languages for hardware design

Kiwi [29] and Liquid metal [38] are two hardware description languages that are extended from Java and C#, respectively. Both of these projects added parallel constructs to the base modern language to design and implement hardware. Chisel [4] is a hardware description language[2] built on top of Scala [64].

3.2 Programming model overview

A design in Gorilla++ consists of design components that are either primitive or composite. Each primitive component corresponds to one dataflow node in the design's GDF model. The primitive components are either programmable engines or predefined infrastructure components. A Gorilla++ program consists of C-kernels associated with the programmable engines and a top-level code that (i) instantiates the primitive components and (ii) composes larger components from smaller ones using composition functions (Section 3.4.2). Figure 3.1 shows the class hierarchy of Gorilla++ components.

We use a simplified histogram accelerator as an example design for introducing Gorilla++ language constructs. In Section 3.5, a more detailed version of this accelerator as the case study for Gorilla++ language is presented. Figure 3.2 shows the engine code and Figure 3.3 shows the top-level code of this simplified histogram accelerator. The accelerator receives input data elements, classifies them to find the corresponding bucket identifiers, and

[2] In order to emphasis the capabilities for designing parameterized hardware, the authors of Chisel call this language a hardware construction language.

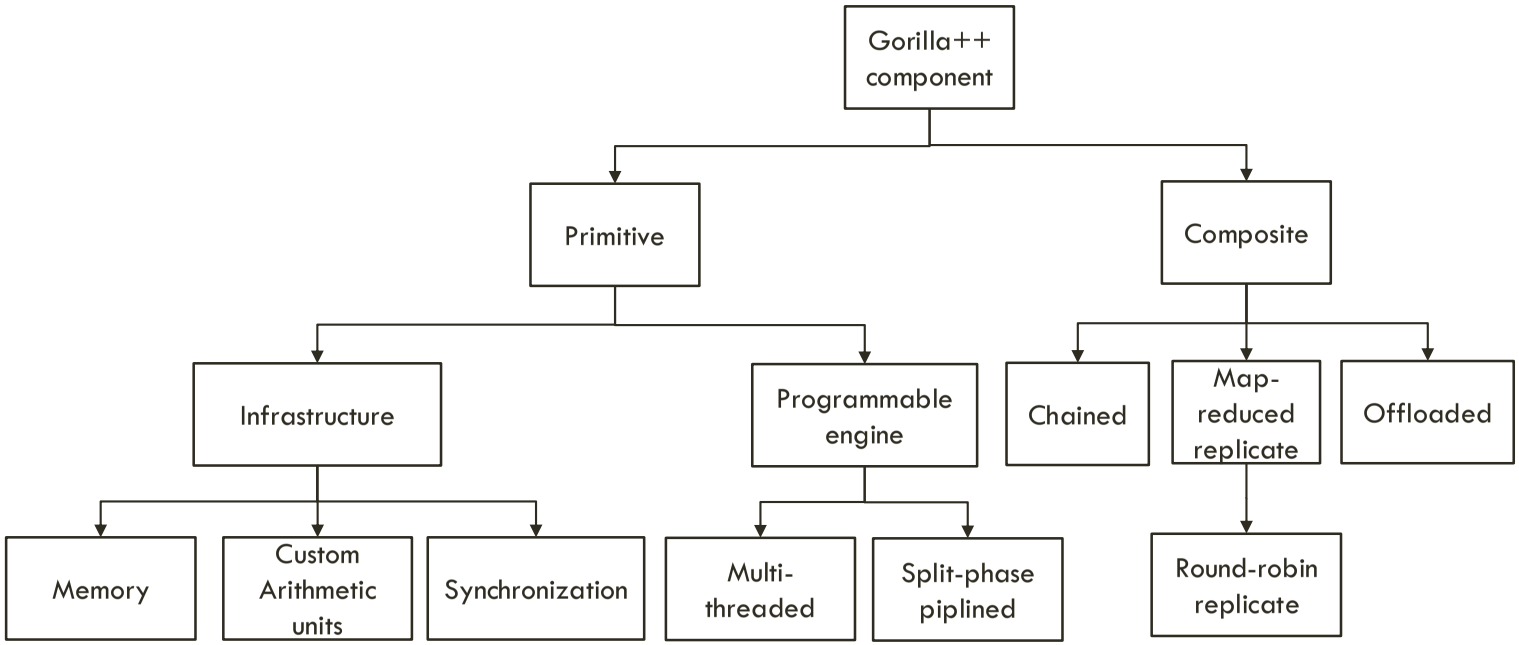


Figure 3.1: The class hierarchy of Gorilla++ components.

```
#pragma INPUT histogramIn_t
#pragma OUTPUT histogramOut_t
#pragma OFFLOAD (memory, memOpIn_t, memOpOut_t)
#define bucketBase 0x1000
uint32_t bucketId;
float bucketValue;

//The code for initialization and reporting
//is not shown.

Receive() {
  bucketId = classify(Input.data);
  bucketValue = memory(READ, bucketBase+bucketId);
  State = Increment;}

Increment() {
  bucketValue++;
  memory(WRITE, bucketBase+bucketId, bucketValue);
  finishNoEmit();}
```

Figure 3.2: The simplified histogram engine code.

increments the bucket values which are stored in a memory component.

```
class Top extends gComponent with include {
 val memory = spMem(1024, 32)
 val histogramEngine = Engine("histogramEngine.c")
 val result = Offload(histogramEngine, memory)
}
```

Figure 3.3: The simplified histogram top-level code.

3.3 Engine code

The engine language is stylized C that specifies the engine interfaces as well as a scheduled description of the computation kernel that is applied on each input data element. The language is designed such that the engines' code can be generated by a standard HLS scheduling front-end from an unscheduled C description. However, when the quality of the scheduling is critical to the performance, the computation can be scheduled manually and written in the form of the Gorilla++ engine language. Appendix A presents the Gorilla++ engine language Backus-Naur form(BNF).

3.3.1 Interface specification

An engine's interface types are specified using C pragmas at the beginning of the engine code including one input declaration, one output declaration, and zero or more offload declarations. The simple histogram engine code has **histogramIn_t** as the input interface type and **histogramOut_t** as the output interface type (see Section 3.5 for details). The engine code declares one offload interface to the memory that keeps the histogram bucket values. The request side of the memory interface has the **memOpIn_t** type and the reply side of the interface has the **memOpOut_t** type.

3.3.2 Processing steps

An engine code consists of one or more processing step(s). Each processing step is defined using a C function and can perform arbitrary arithmetic/logic computation. It can also call other offloaded components. For each call, the engine generates/receives request/response messages to/from the offloaded components through the engine's offload interfaces. In addition, each step explicitly specifies the next step(s) based on the result of computation and/or the responses from the offloaded components. Note that the offloaded components themselves may be user-defined engines and can be generated by the Gorilla++ compiler. In the simplified histogram example the engine has two processing steps, **Receive** and **Increment**. Both of these steps call **memory** as an offloaded component.

3.3.3 Data types

Apart from standard C data types, Gorilla++ supports data types with specified widths in the form of the **uint** data type. For example, **uint48_t** specifies a 48 bits unsigned integer.

Bundled types are specified using C structures. The support of bundled data types is specially useful for processing protocol fields and complex data structures. In the simplified histogram engine, all interface types are C structures.

Type casting can be used to convert a variable with a given data type to another data type. Casting one bundle type to another results into flattening the source bundle to a bit stream and remapping the bit stream into the target bundle type.

3.3.4 Predication

The programmer can use conditional statements in each processing step. The conditional statements are translated into predication logic by the compiler.

3.3.5 Transitioning between processing steps

Transitioning between processing steps is done by explicit assignments to the **State** variable. At the end of each processing step, the Gorilla++ infrastructure switches to the processing step that is specified by this variable. The **Receive** step in the simple histogram engine specifies the next processing step (**Increment**) by assigning its value to the built-in **State** variable.

3.3.6 Receive/send operations

The input data element is accessible through the **Input** variable. The outputs should be assigned to the **Output** variable. Receiving an input is done implicitly when the engine restarts in the first processing step. An output is sent by calling either the **finish** function or the **emit** function. The difference between the **finish** and **emit** is that **finish** returns to the first processing step of the engine, but emit returns to the processing step specified as its argument. The programmer can also use **finishNoEmit** to restart the state machine without emitting any data element to the output. The simplified histogram engine reads the input data values from **Input** variable in the **Receive** processing step. It uses **finishNoEmit** to restart the kernel and start processing of the next data element.

3.4 Top-level code

The top-level code of a design is written in Chisel [4]. In the top-level code, a programmer instantiates a primitive component by calling the corresponding function. The function returns the generated engine or infrastructure component, which can be used later for composition. A composition function accepts one or more components and returns the composed component that encloses the argument components.

3.4.1 Primitive component instantiation

In order to instantiate a programmable engine from a C code, the following functions can be used. The **Engine** function generates an engine from the argument C code that can process a single input data element at any given time. The **MTEngine** and **PipeEngine** functions, however, generate multi-threaded and pipelined engines respectively. They can process multiple input data elements concurrently. A multi-threaded engine can overlap the offload times associated to different data elements. A pipelined engine, however, can overlap both offload times and computation times associated to different data elements. Section 4.4 describes more details on the difference between these three engine micro-architectures.

```
//simple engine
val e1 = Engine("engineKernel.c")

//multi-threaded engine
val e2 = MTEngine("engineKerenel.c", numOfThreads=2)
```

```
//pipelined engine
val e3 = PipeEngine("engineKernel.c")
val e4 = PipeEngine("engineKernel.c", numOfThreads=2)
```

Similarly, the following functions can be used to instantiate infrastructure components including lock components, floating-point arithmetic units, and scratch-pad memories.

```
//lock engine
val l = lock(numOfLocks=128)

//scratch-pad memory
val m = spMem(height=1024, width=32)

//single-precision floating point operations
val add = fpAdder()
val sub = fpSubtractor()
val mul = fpMultiplier()
val div = fpDivider()
val sqrt = fpSqrt()

//double-precision floating point operations
val add = fpdpAdder()
```

```
val sub = fpdpSubtractor()
val mul = fpdpMultiplier()
val div = fpdpDivider()
val sqrt = fpdpSqrt()
```

A **lock** component is used for synchronization between different execution contexts. When an engine needs to own a lock, it calls the **lock** function with the lock id as the argument. Calling the **lock** function is treated as any other offload call. Therefore, a request will be sent to the **lock** component. The **lock** component receives the request and if no other execution context owns the lock, it replies immediately to the requester engine – acknowledging that the requester owns the lock. It also sets the requester as the last owner of the lock in the **last serviced** memory. If any other requester sends a request to own the same lock, the bit associated to its thread id will be set in the corresponding **wait-list bit-vector** entry and no acknowledgement is sent to the requester. When the owner releases a lock, the **lock** component (i) sends the release acknowledgement to the owner, (ii) selects one of the waiters, if any, as the next owner of the lock, and (iii) sends the new owner the lock acquire acknowledgement. In order to avoid starvation, a fair arbiter is used to select the next owner between the waiters for the lock. The internal micro-architecture of a lock component is shown in Figure 3.4.

In the top-level code of the simplified histogram accelerator, the histogram engine is generated by calling the **Engine** function and a scratch-pad is generated using the **spMem** function.

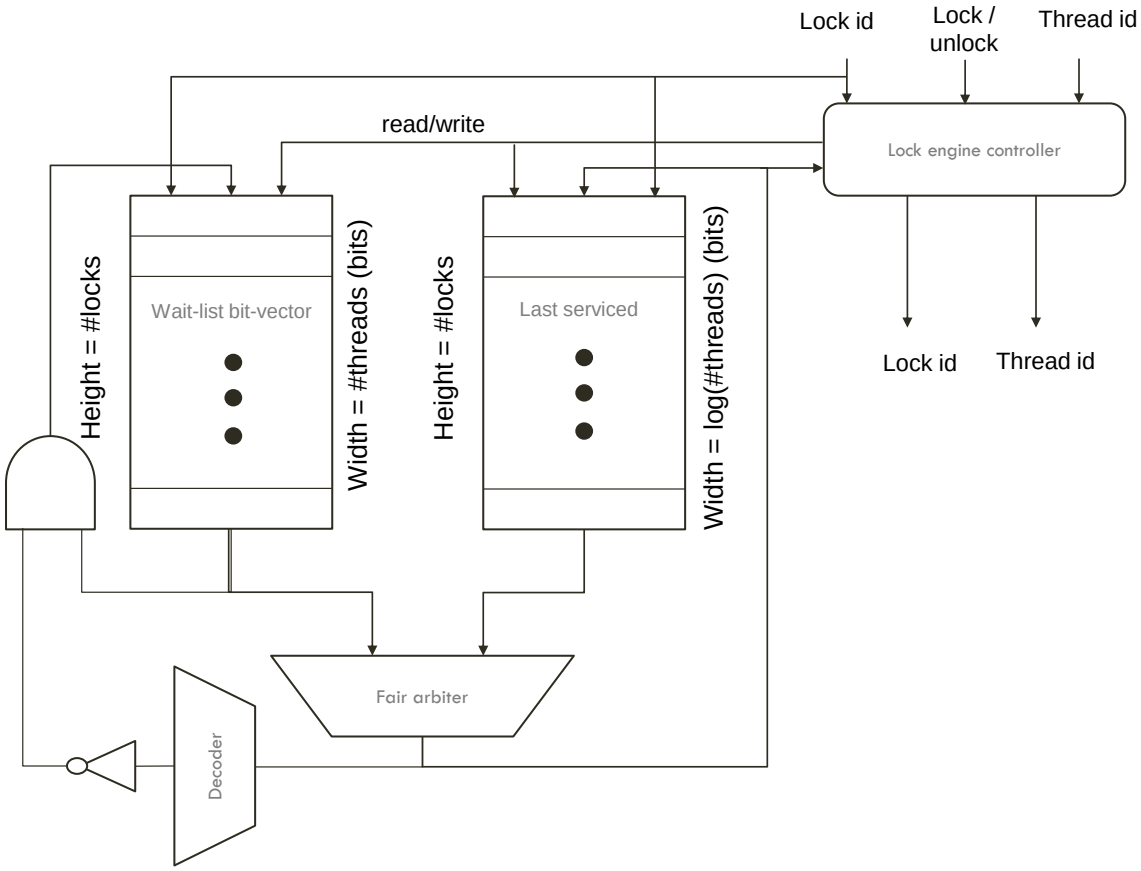


Figure 3.4: Architecture of the Gorilla++ lock engine.

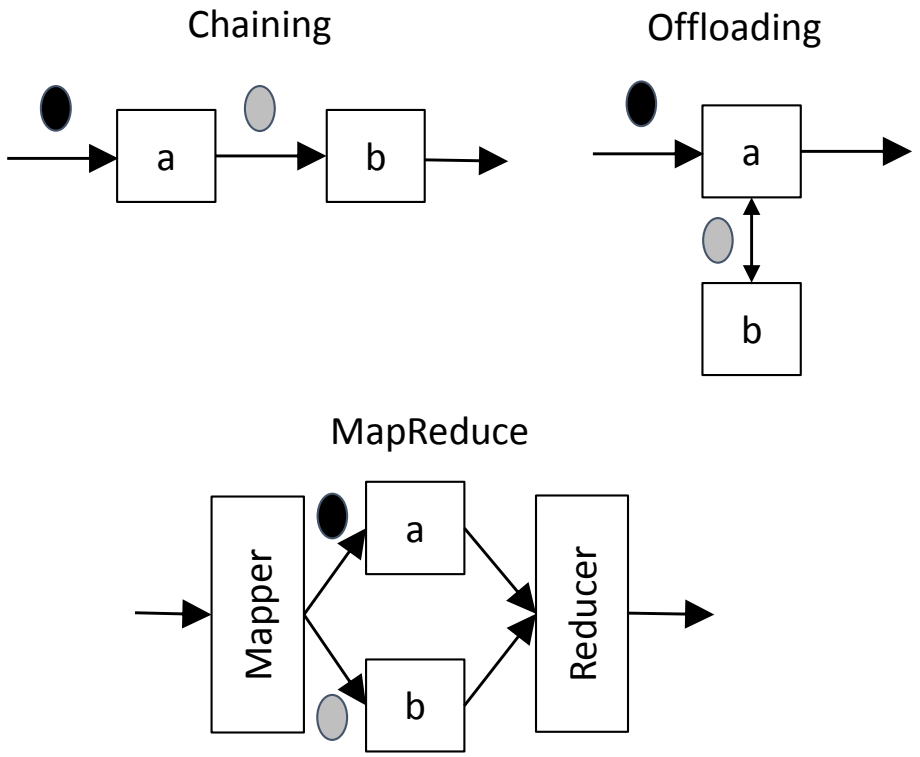


Figure 3.5: Chain, Offload, and MapReduce as the Gorilla++ composition templates.

3.4.2 Composition functions

Figure 3.5 shows different composition templates in Gorilla++. Each template can be applied on argument components to generate a new component by calling the corresponding composition function. The **Chain** composition function creates a coarse-grain pipeline that connects the first component's output to the second component's input.

The **Offload** composition function is used to connect request/reply interfaces of an offloading component to the input/output interfaces of the offloaded one. Offloaded components can be shared between different offloading

components, e.g., a memory shared between two engines.

The **Replicate** composition function creates multiple instances of a component. In its generic form, i.e., **Map-reduced Replicate** can accept user-defined mapper and reducer functions to specify the way data elements should be distributed to and gathered from the replicated components. If the user does not specify the functions, the default, fair round-robin functions are used.

Note that the argument components to the composition functions can be generated from either instantiation of primitive components or calling other composition functions. The result of the last composition function should be assigned to a special variable **result**. The following examples show the syntax of using different composition functions.

In the top-level code of the simplified histogram example, **Offload** composition function is used to attach the histogram engine to a scratch-pad memory.

```
//Chain composition
val chainedComponent1 = Chain(component1, component2)
val chainedComponent2 = Chain(component1, component2,
 component3)

//Offload composition
val offloadComponent = Offload(component1, component2,
 portName="component2Port")
```

```
//Round-robin replicate composition
val replicatedcomponent1 = Replicate(component,
 replicationFactor=2)
//Map-reduced replicate composition
val replicatedcomponent2 = Replicate(component,
 replicationFactor=2, mapperFunction, reducerFunction)
```

3.5 Case study: histogram accelerator

A more enhanced version of the histogram accelerator is used as a case study for Gorilla++. This case study helps to cover more detailed aspects of the Gorilla++ language. Unlike the simplified histogram accelerator, this accelerator uses two histogram engines to improve the throughput and includes the components for generating the inputs and reporting the outputs. It also keeps the bucket values in a DRAM instead of a scratch-pad memory. Since multiple engines access bucket values concurrently, a lock component is used to provide mutual exclusion.

Figure 3.6 shows the micro-architecture of the histogram accelerator and Figure 3.7 shows the corresponding top-level code of the accelerator. Figure 3.8 shows the micro-architecture of a non-blocking cache that is used in the histogram accelerator to facilitate the accelerator's accesses to a DRAM. Figure 3.9 shows the corresponding top-level code of the cache. Figures 3.10 and 3.11 show the C-code associated with two main engines in the accelerator and cache: the histogram engine and the cache controller. The input generator in the accelerator streams out input numbers that are distributed between two

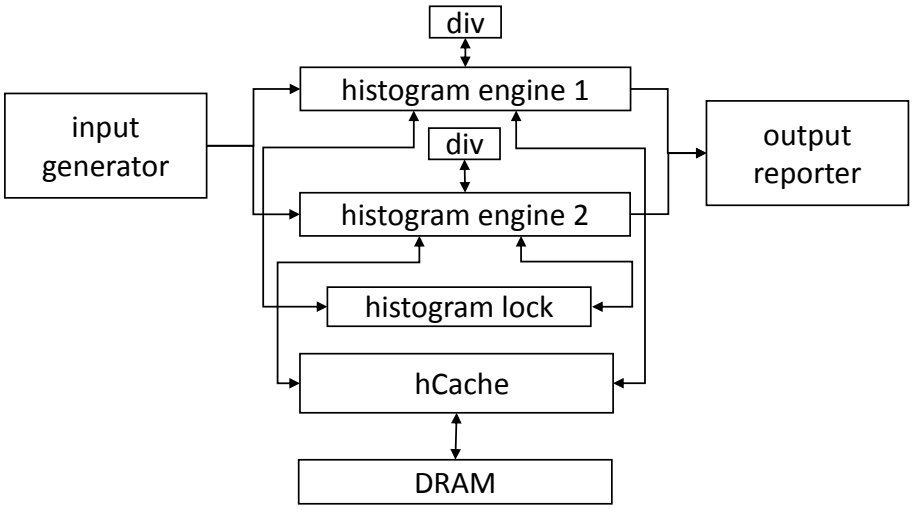


Figure 3.6: The micro-architecture of a histogram accelerator.

```
class Top extends gComponent with include {
 val inputGenerator = Engine("inputGenerator.c")
 val histogramEngine = Engine("histogramEngine.c")
 val outputReporter = Engine("outputReporter.c")
 val hCache = Cache(height=1024, lineSize=128, tagSize=20, threads=1)
 val memory = Offlod(hCache, DRAM)
 val histogramLock = lock(BUCKETS)
 val div = FPDivider()
 val histograms = Replicate(Offload(histogramEngine, div), 2)
 val histogramsAndIO= Chain(inputGenerator, histograms, outputReporter)
 val result = Offload(histogramAndIO, memory, histogramLock)
}
```

Figure 3.7: The top-level code of the histogram accelerator.

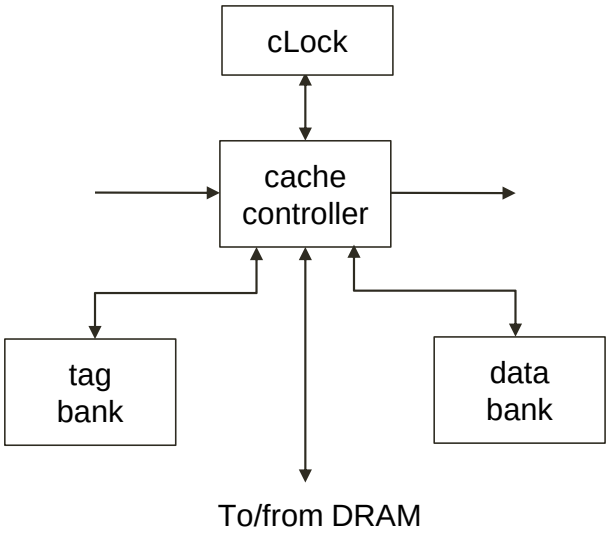


Figure 3.8: The micro-architecture of a hCache.

```
class Cache(height, lineSize, tagSize, threads: Int)
 extends gComponent with include {
 val cacheController = MTEngine("cacheController.c", threads)
 val dataBank = spMem(height, lineSize)
 val tagBank = spMem(height, tagSize)
 val cacheLock = lock(height)
 val result = Offload(cacheController, dataBank, tagBank, cacheLock)
}
```

Figure 3.9: The top-level code of the hCache.

```
#pragma INPUT histogramIn_t
#pragam OUTPUT histogramOut_t
#pragma OFFLOAD (memory, memOpIn_t, MemOpOut_t)
#pragma OFFLOAD (hLock, lockIn_t, lockOut_t)
#pragma OFFLOAD (div, fuIn_t, fuOut_t)
#pragma CONCURRENT_SAFE
#define bucketBase 0x1000
uint32_t bucketId;
float bucketValue;

Receive() {
 if (Input.command == INITIALIZE) {
  bucketId = 0;
  State = Intialize; }
 if (Input.command == CALCULATE) {
  bucketId = classify(Input.data);
  hLock(ACQUIRE, bucketId);
  bucketValue = memory(READ, bucketBase+bucketId);
  State = Increment; }
 if (Input.command == REPORT) {
  bucketId = 0; State = Report; } }
Initialize() {
 if (bucketId == NUM_OF_BUCKETS) {
   finishNoEmit(); } else {
   memory(WRITE, bucketBase+bucketId, 0);
   buckeId++; State = Initialize; } }
Increment() {
  bucketValue++;
  memory(WRITE, bucketBase+bucketId, bucketValue);
  hLock(RELEASE, bucketId); finishNoEmit(); }
Report() {
 if (bucketId == NUM_OF_BUCKETS) {
   Output.status = DONE;
   finish(); } else {
   Output.status = REPORT_COUNTER
   Output.counter = memory(READ, bucketBase+bucketId);
   buckeId++; Emit(); } }
```

Figure 3.10: The histogram engine code.

```
#pragma INPUT memOpIn_t
#pragma OUTPUT memOpOut_t
#pragma OFFLOAD (tag, memReqCacheTagBank_t, memRepCacheTagBank_t)
#pragma OFFLOAD (data, memReqCacheDataBank_t, memRepCacheDataBank_t)
#pragma OFFLOAD (dram, memReqDram_t, memRepDram_t)
#CONCURRENT_SAFE
uint16_t index;
cacheLine_t cacheLine;
tag_t tag;

FirstAndLoadHit() {
 index = INDEX(Input.addr);
 cacheLock(index, ACQUIRE);
 cacheLine = dataBank(READ, index);
 tag = tagBank(READ, index);
 if ((State = ACTION(tag, Input)) == LOADHIT) {
  Output.data = WORD(cacheLine, Input.addr);
  finish(); } }
StoreHit() {
 tagBank(WRITE, index, FRESHTAG(Input.addr));
 dataBank(WRITE, index, cacheLine);
 finish(); }
WriteBack() {
 dram(WRITE, tag.addr, cacheLine);
 State = CacheFill; }
CacheFill() {
 cacheLine = dram(READ, Input.addr);
 if (Input.command == LOAD) {
   Output.data = WORD(cacheLine, Input.addr); } else { //STORE
  cacheLine = writeWordInLine(cacheLine, Input.data); }
 tagBank(WRITE, index, FRESHTAG(Input.addr));
 dataBank(WRITE, index, cacheLine);
 cacheLock(RELEASE, index);
 finish(); }
```

Figure 3.11: The cache controller engine code.

histogram engines.[3]

The input bundle of the histogram carries a floating point **data** field, the data that should be processed by the histogram algorithm. It also includes **initialize** or **finish** fields to determine whether the input data element is for initialization, the end of the input values, or neither. Similarly, the output interface has a bundle type to indicate whether the data element is carrying one of the counters associated with a bucket or it is a message indicating that the histogram counters are finished.

```
typedef struct {
  command_t command;
  float data;
} histogramIn_t;

typedef struct {
  status_t status;
  uint64_t counter;
} histogramIn_t;
```

The histogram engine also defines the offload interfaces, including the memory interface, lock interface, and floating point interface, to access the corresponding components.

[3]Multiple engines are used to showcase all Gorilla++ composition functions. However, even if the programmer specifies a single engine, Gorilla++ generates the required number of engines automatically.

The engine has three computation phases for (i) initialization, (ii) computation, and (iii) reporting. In the initialization phase, the accelerator resets all the histogram counters to zero. In the computation phase, the engines classify each number and increment the corresponding bucket value. The bucket values are kept in the DRAM. A shared infrastructure lock component is used to guarantee the mutual exclusion of accesses to the bucket values. At the end, a histogram engine switches to the reporting phase and reads and streams out the bucket values, which are received by the reporter component. The engine consists of four processing steps. The first processing step, **Receive**, receives an incoming data element and jumps to one of the other three processing steps, **Initialize**, **Increment**, or **Report**, based on the value of the current computation phase.

In the initialization and computation phases, the accelerator finishes the processing of an input, by calling the **finishNoEmit** function to restart the state machine. The **emit** function is used to report individual counter values and the **finish** is used when the reporting of histogram counters is completed.

In the histogram top-level code, **Engine** function is used to instantiate input generator, histogram engine, and output reporter. The input generator, the histogram engine, and the output generator are chained together in a back-to-back fashion using **Chain** composition function. **Offload** is used in the accelerator to connect memory sub-system, lock engine, and floating point divider to the histogram engines. **Replicate** is used in the accelerator to create two instances of the histogram engine and improve the throughput. Histogram engine uses **lock** for synchronization. It also uses a **fpDivider** (floating point divider) in the classify function.

Like a histogram that receives a stream of input numbers, a cache is also designed as a piece of hardware that processes a stream of memory operations. The cache controller receives the memory requests and uses offloaded data-bank, tag-bank, and DRAM to service the memory requests. The **FirstAndLoadHit** processing step reads the data-bank and tag-bank and service the hit load requests immediately. It switches to **StoreHit** processing step in the case of a store hit. When the request is a miss request and the current cache line is dirty, **FirstAndLoadHit** switches to **WriteBack** processing step. The miss requests are serviced in the **CacheFill** processing steps.

When the cache controller is compiled as a multi-threaded engine, a non-blocking cache that can service concurrent memory requests is generated. Note that the concurrent memory requests are associated with different histogram inputs. Since data and tag banks are offloaded, they are shared between the threads. A lock component in the cache is used to provide the mutual exclusion of accesses to each cache line. Therefore, if a thread in the cache controller starts processing a memory operation on a cache line, consequent memory operations on the same line are stalled until processing of the first request is finished.

3.6 Comparing Gorilla++ language with closely related work

Similar to SIMT-based programming model, Gorilla++ uses a data-parallel programming model with no default ordering constraint. However, it is built around a dataflow-centric execution model and uses structured composition. These two characteristics enable automatic refinement of the micro-architecture.

Gorilla++ composition functions are similar to Lava combinators. However, they are used only for coarse-grain composition. Gorilla++ uses sequential C programming model to specify the node computations. Unlike Lava, Gorilla++ does not support recursive combinators.

Similar to StreamIt [85], Gorilla++ is using structured composition of streaming components. Gorilla++, however, uses offloading as the first-order construct to support accesses to global memory and also uses lock-based synchronization to provide data-race freedom.

Chapter 4

Gorilla++ compiler

This chapter describes the Gorilla++ compiler which generates a gate-level description of the accelerator from a high-level Gorilla++ code. The compiler can apply a set of specified refinements on the micro-architecture of the generated accelerator.

4.1 Related work: generating and refining parallel micro-architectures in high-level synthesis

The related work of Gorilla++ compiler is presented in two different categories. The first category addresses the problem of loop pipelining in a sequential programming model like C. The main focus in this category is to schedule the operations in different pipeline stages without violating the data dependencies between operations in a single loop iteration or between operations in different loop iterations. Gorilla++, however, preserves the dependencies between operations in a different fashion. It makes sure that different stages of the pipeline (either a data-path pipeline in a pipelined engine or an execution pipeline in a multi-threaded engine (Section 4.4)) executes operations associated to different data elements. The second category addresses the problem of refining a micro-architecture to improve its performance or reduce the hardware area.

4.1.1 Loop-pipelining in HLS

Loop winding [26], rotation scheduling [14], and time constraint loop pipelining [74] are among the early techniques for generating pipelines in HLS. Iterative modulo scheduling, a method that was originally built for scheduling instructions for very large instruction width machines, is used in many HLS tools including PICO [77], C-to-Verilog [9], and Legup [11] for generating pipelines. More recent work [10, 90] has addressed the problem of modulo scheduling using a system of difference constraints equations. Nurvitadhi et al. [63] presented a method to generate multi-threaded in-order pipelines from a transactional input description.

Liu et al. [57] proposed to split loop bodies into two coarse-grain pipelines that work independently. The first stage, which needs to be sequential due to data dependencies, feeds the second stage, which can be parallel.

4.1.2 Refinement of rule-based designs

Lis and Arvind invented an algorithm [56] for safely composing rules in a rule-based design. They used this rule composition technique and invented a dot-product transformation that can automatically generate a multi-issue micro-architecture specification of a processor from a single-issue specification. While multi-issue can fetch, decode, and execute multiple instructions in a given cycle, it can also preserve the sequential execution semantic between the instructions.

4.1.3 Refinement of statically allocated functional designs

SAFL [61] proposed the refinement of micro-architectures by rewriting their functional descriptions. For example, if two consumer functions use the

same producer function, by default, the producer is implemented as a shared hardware between the consumers. However, a source-level transformation can change the design to have two different instances of the same producer function; so, each of the consumers has a dedicated producer component. This can be done to improve the parallelism in and consequently the performance of the design.

4.1.4 Refinement of SDF-based designs

An important feature in StreamIt [85] project is the ability to reform the SDF graph to achieve a given performance target for a set of resource constraints. Hagiescu et al. and Hormati et al. [30, 37] extended the same concept for refining hardware micro-architectures using streaming refinements, e.g., replication refinement. Cong et al. [20] added pipelining refinement and Li et al. [54] added merging and splitting refinements to this method.

4.2 Compiler flow

The input to the Gorilla++ compiler consists of:

- An application code including:
  - A top-level composition code that determines the design's base micro-architecture.
  - Scheduled engines' code.
- A set of micro-architecture refinements, which are applied to the design's base micro-architecture.

The compiler translates the application code into a Chisel description of an accelerator by combining the functionality of the engines with the refined micro-architecture of the accelerator. The refined micro-architecture is the result of refining the base micro-architecture using the specified refinements.

The engine compiler translates the engines' code to Chisel code based on the specified micro-architecture template. The engine compiler also generates interface information, which is required to infer the interface types of the composite components. An engine's code can be in form of scheduled, stylized C, described in Chapter 3. This is preferable when the programmer wants strict control over the generated states in the engine's state machine or the pipeline stages in the engine's pipeline. Alternatively, an HLS front end can be used to translate an unscheduled C code to a scheduled intermediate form usable by the Gorilla++ engine compiler. For the benchmarks of this thesis, the first approach is used. Note that Gorilla++ refinements are back end optimizations and are independent of the standard HLS scheduling techniques.

The refiner module applies the input set of refinements to the composition code and generates a new composition code.[1] The collection of the generated engines and top-level Chisel code is passed to the Chisel compiler to generate either a gate-level C-simulation model or a gate-level Verilog model.

[1]The engines' code solely describes the functionality of the engines and it is completely independent of their micro-architectures. Micro-architecture details, however, are encoded in the composition code.

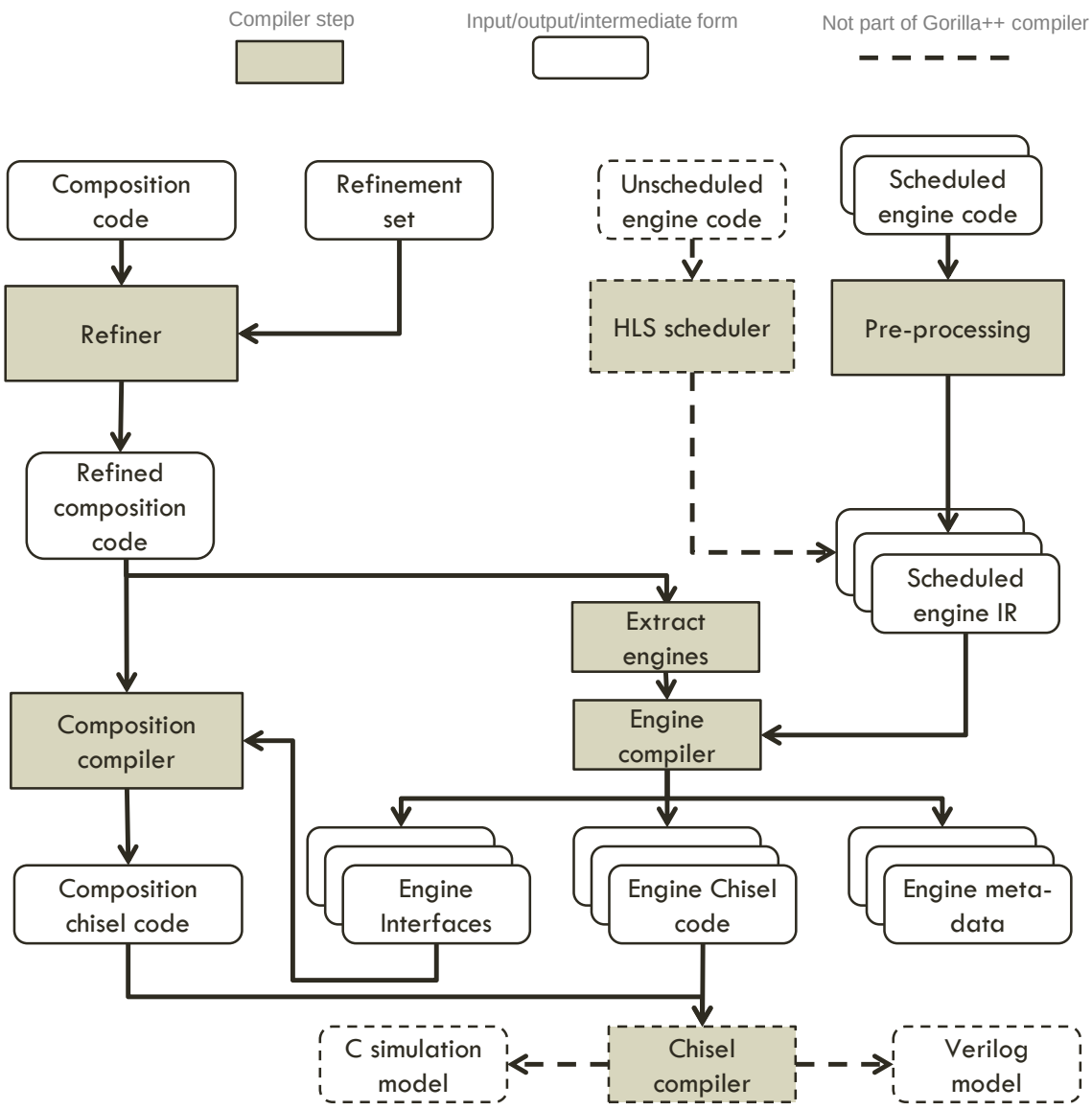


Figure 4.1: Gorilla++ compiler flow-chart.

4.3 Composition compiler

Figure 4.2 shows the composite components of the histogram accelerator shown in Section 3.5 of Chapter 3. The **histograms** component is the result of replication of two instances of the **histogramEngine**; each uses an offloaded **div** floating point component. The **histogramsAndIO** is the result of chaining three components, **inputGenerator**, **histograms**, and **outputReporter**. The top-level **result** component is generated by connecting the **histogramsAndIO** to **histogram lock** and **memory** components using offload composition. The **memory** component itself is generated by connecting **hCache** to the **DRAM** using offload composition.

4.3.1 Lazy creation of components

The composition functions create components lazily using component generators, which are in form of thunk subroutines [86]. A composition function receives a meta data object for each argument component and returns a meta data object for the composite component. The meta data class is shown below.

```
class componentMetaData {
  inData: () => Data //input interface generator
  outData: () => Data //output interface generator
  offData: ArrayBuffer[offData] //offload interface generators
  generator: () => gComponent //component generator
}
class gComponent {
```

```
  //Only the major variables are shown
  val name: String
  val lhsName: String
  val parent: gComponent
  val children: ArrayBuffer[gComponent]
  val nodeType: Int
  val engineAttributes: EngineAttributes
  val pc : PC
  val concurrentSafe : Boolean
}
class offData {
  interfaceName: String //offload interface name
  reqData: () => Data //offload request interface generator
  repData: () => Data //offload reply interface generator
}
```

It consists of (i) the generators for input, output, and offload interfaces and (ii) the generator of the component itself. Based on this information, the function derives the composite component's input, output, and offload interface types and creates a generator for the composite component that (i) calls the argument components' generators to instantiate them, (ii) connects the argument components and composite component interfaces according to the semantic of the function, and (iii) connects the performance counter rings between the argument components together and to the composite components' corresponding ports.

Figures 4.3, 4.4, and 4.5 show these detailed actions taken by the **Chain**, **Offload**, and **Replicate** composition functions. In the Chain composition function, which has two argument components, the input interface of the generated component is connected to the first component's input and its output is connected to the second component's output. For the offload interfaces of the argument components that have the same port name, an arbiter/dispatcher logic is instantiated automatically to merge the interfaces and connect them to the corresponding interface in the composite component. The rest of the offload interfaces are connected directly to the composite component's interfaces.

The input/output interfaces of a composite component generated by the **Offload** are connected to the input/output interfaces of the first argument component (offloading component). The set of offload interfaces in the composite component are the union of the two argument components' offload sets minus the argument offload interface. Offload assumes that its argument components, the offloading and offloaded components, do not have any common offload port.

For a composite component generated by the **Replicate**, the necessary arbitration/distribution logic is added to dispatch the input data elements to different instances of the argument component (replicated component) and aggregate the output data elements from them. Also, the necessary arbitration logic is automatically added to each offload interface to aggregate requests from different instances of the replicated component to send them to a single offload interface in the composite component. Similarly, the necessary distribution logic is added to dispatch the replies from the composite component's offload interface to the replicated instances of the argument component.

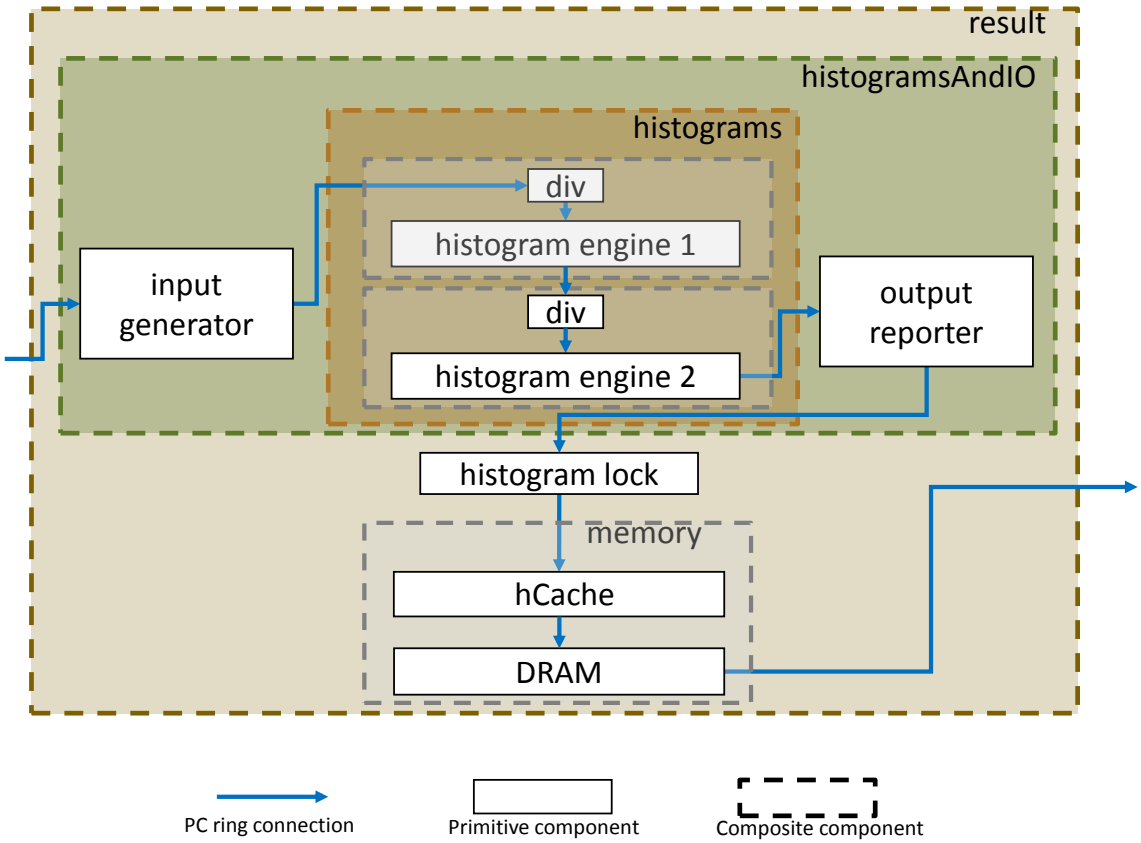


Figure 4.2: The composition components and performance counter ring in the histogram accelerator.

Operation	In	Out	Offloads	pcRing

z=
Chain(x,y)
	z.in
	<>
	x.in
		z.out
		<>
		y.out
			foreach offload w in (x.offloads ∩ y.offloads) {
			z.offloads.w.req <> arbiter(x.offloads.w.req, y.offloads.w.req)
			(x.offloads.w.rep, y.offloads.w.rep) <> dispatcher(z.offloads.w.rep)
			}
			
			foreach offload w in ((x.offloads – (x.offloads ∩ y.offloads)) {
			x.offloads.w <> z.offloads.w
			}
				z.pcRing.in <>
				x.pcRing.in
				
				z.pcRing.out <>
				y.pcRing.in
				
				z.pcRing.out <>
				y.pcRing.out

Figure 4.3: The pseudo code for the **Chain** composition function.[1]

Operation	In	Out	Offloads	pcRing

z=
Offload(x,y,p)
	z.input
	<>
	x.input
		z.output
		<>
		x.output
			foreach offload w in (x.offloads – p) {
			x.w.req <> z.w.req
			x.w.rep <> z.w.rep
			}
			foreach offload w in (y.offloads) {
			y.w.req <> z.w.req
			y.w.rep <> z.w.rep
			}
				z.pcRing.in <>
				x.pcRing.in
				
				z.pcRing.out <>
				y.pcRing.in
				
				z.pcRing.out <>
				y.pcRing.out

Figure 4.4: The pseudo code for the **Offload** composition function.[1]

Operation	In	Out	Offloads	pcRing
z= Replicate(x,n)	(x(1).in, …, x(n).in) <> dispatch(z.in)	z.out = arbiter(x(1).out, …, x(n).out)	foreach offload w in x.offloads { z.offloads.w.req <> arbiter(x(1).offloads.w.req, …, x(n).offloads.w.req) (x(1).offloads.w.rep, …, x(n).offloads.w.rep) <> dispatcher(z.offloads.w.rep) }	z.pcRing.in <> x(1).pcRing.in z.pcRing.out <> x(n).pcRing.out for (I in 1 to n-1) { x(i).pcRing.out <> x(i+1).pcRing.in }

Figure 4.5: The pseudo code for the **Replicate** composition function.[1]

4.3.2 Generating the performance counter ring

Composition functions also create a performance counter ring between the components. The ring is used to send commands to the logic that controls performance counters and is also used to read the values of the counters. Figure 4.2 shows the performance counter ring for the histogram accelerator.

Each component has an input ring port and an output ring port. It receives tokens from its input ring port and either (i) replies to the token if the token is addressed to the component itself, or (ii) passes the token to the output ring port. In a composite component with a certain number of argument components, the output ring port of each argument component is connected to the next argument component's input ring port. The first argument component's input ring port is connected to the composite component's input ring port and the last argument component's output ring port is connected to the

composite component's output ring port.

4.4 Engine compiler

The Gorilla++ engine compiler uses templates to compile scheduled processing steps into the hardware. A template is designed for an arbitrary number of offload interfaces, an arbitrary number of processing steps, arbitrary functionality in processing steps, and arbitrary interface types. Gorilla++'s three engine templates are (i) simple state machine engines, (ii) multi-threaded engines, and (iii) pipelined engines.

4.4.1 Simple state machine engines

When an engine is translated into a simple state machine, all processing steps that do not have any offload calls are translated into a single state in the state machine and all processing steps that have offload calls are translated into two states in the state machine, a state for computations before offload call(s) and a state for computations after offload call(s). Global variables are translated into context memories and variables that are local to a processing steps are translated into wires. Figure 4.6 shows an implemented engine using a simple state machine template. Offload calls are shown using dashed lines between the processing steps and the offloaded components.

4.4.2 Multi-threaded engines

Whenever an engine is spending a large amount of time in the offload call(s), it is reasonable to switch to the processing of another input data el-

[1]The <> operator is a bulk connectivity operator used in Chisel.

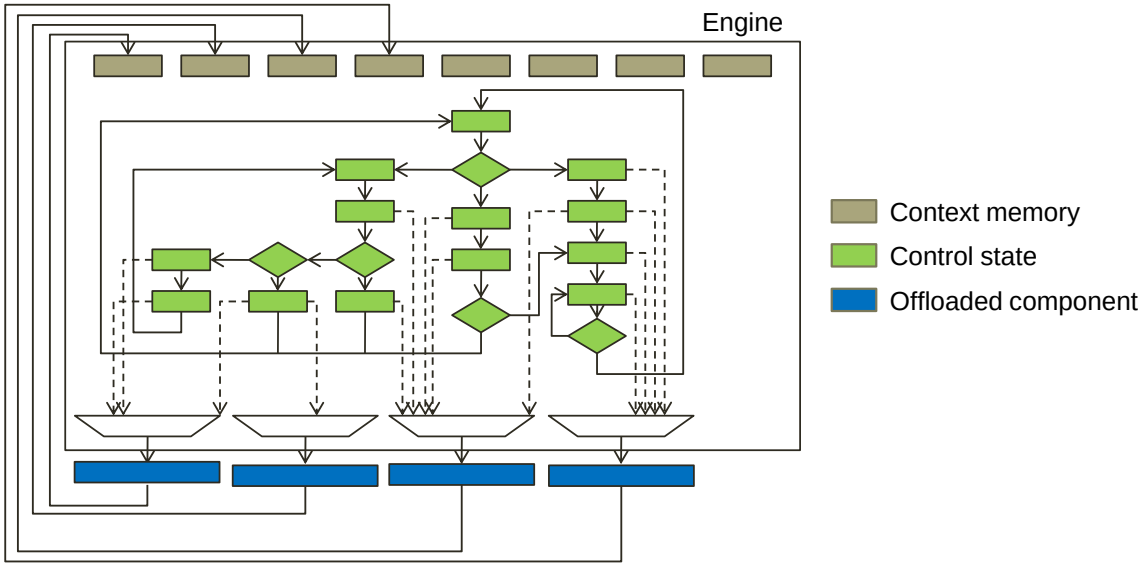


Figure 4.6: A simple state machine engine.

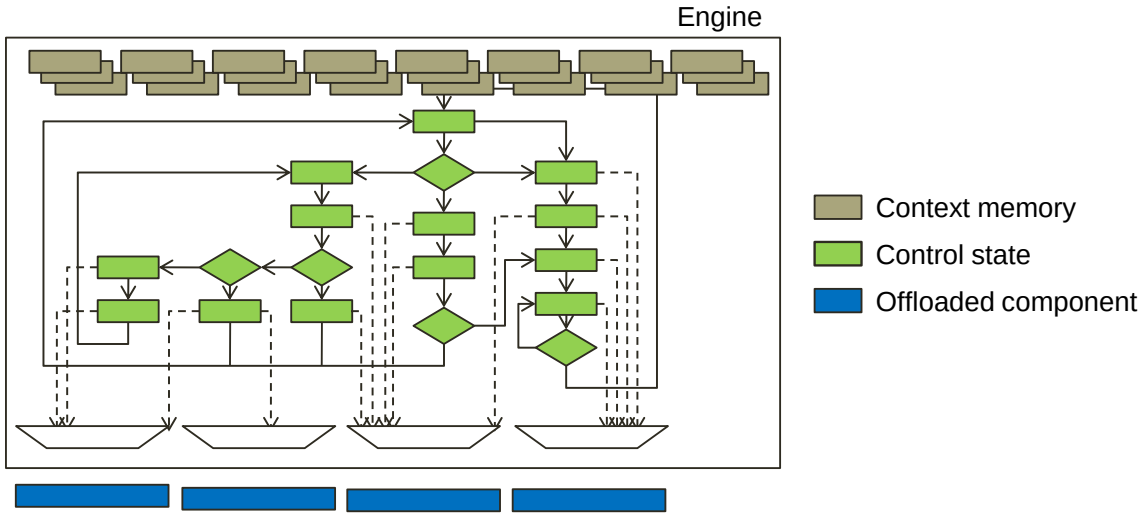


Figure 4.7: The logical view of a multi-threaded engine.

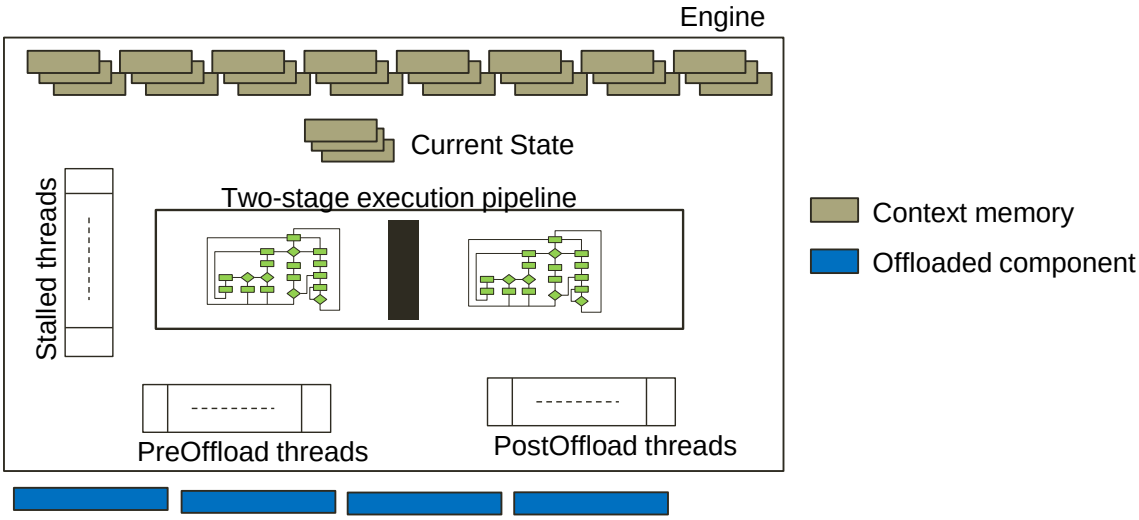


Figure 4.8: Implementation of a multi-threaded engine using a two-stage execution pipeline.

ement while waiting for the response of the offload call(s). This is possible because of the data-parallel nature of the Gorilla++ target applications. Inspired by multi-threading technique in high-performance processors [23], we refer to this type of concurrency as multi-threading. Each thread in an engine is responsible for processing an input data element. Whenever a thread calls an offload, the engine switches to another thread, processing another input data element. In a multi-threaded engine, the necessary data-path and control logic is shared between different threads. However, the engine needs to have a separate set of context memories for each thread (Figure 4.7).

4.4.2.1 Execution pipeline

A multi-threaded engine (shown in Figure 4.8) executes the processing steps using a two-stage execution pipeline. The first stage of the pipeline, the

preOff stage, executes all the computations of a processing step before the offload calls. The second stage, the postOff stage, executes the computations after offload calls. The thread associated with a given data element is either in the preOff or postOff stage of the execution pipeline. Therefore, the preOff and postOff computations of a single input data element are not executed in the same clock cycle and consequently control or data dependencies are preserved.

The engine keeps track of threads using three bit vectors. The first bit vector indicates whether a thread is stalled, waiting for an input data element. The second bit vector contains all the threads that are in the preOff stage, either performing the preOff computations or waiting for the offload request acknowledgements.[2] The third bit vector contains all threads that are in the postOff stage, either performing the postOff computations or waiting for offload replies.

At a given time, it is possible to have more than one candidate thread to move from one bit vector to another. For the stalled bit vector, there might be more than one stalled thread to start the computation of a data element. For the pre-offload bit vector, there might be more than one thread ready to execute the pre-offload computation and send its request(s) to the offloaded component(s). Similarly, for the post-offload bit vector, there might be more than one thread that is finished with the post-offload computation and has also received the offload replies. In all three cases, a fair encoder is used to select a winner thread from the candidate threads. The fair encoders guarantee that the threads are not starved in any of these three stages.

[2]It is possible that offloaded component is not ready to accept a new request from the offloading engine

4.4.2.2 Thread contexts

Thread contexts consist of all global variables in an engine kernel. They also include the implicit variables, including **Input**, **Output**, **State**, and **emitReturnState** (Section 4.4.2.3) variables. Each of these variables is stored in a context memory, which is indexed using thread id of the current active thread.

4.4.2.3 Receive/send processing steps

Each thread in a processing engine has a dedicated processing step for receiving an input data element. Also, it has a dedicated processing step for sending an output data element. These steps are hidden from the programmer.

The free threads stay in the receive processing step until an input data element is assigned to them. When a new data element arrives, a thread from the pool of available threads is selected for processing the data element. The **State** variable for the thread is changed to the first processing step in the computation kernel. The engine asserts the ready signal on its input as long as there is an available free thread in the engine.

When threads emit outputs, they automatically switch to the send processing step. A thread stays in the send processing step until it receives the assertion of transfer using the ready signal of the output port. While staying in this processing step, the engine asserts its valid signal on the output port.

4.4.2.4 Offload dispatch and wakeup logic

Apart from the preOff computations specified in a processing step, the preOff stage is responsible for generating request(s) to the offloaded compo-

nents. The engine leaves the preOff stage and enters the postOff stage whenever all requests for all offload calls in a particular processing step are accepted by the corresponding offloaded component. Also, the engine leaves the postOff stage and enters the preOff stage of the next processing step whenever all replies form the offload calls are received.

When a thread in a preOff stage calls an offloaded component, a request is sent to the component by asserting the valid signal on the corresponding request port. The request is kept asserted until the component acknowledges receiving the request by asserting the ready signal on the same port (see Figure 4.9 for details). The actual request data (not shown in the Figure) is the argument of the offload call. It is sent along the request valid signal. Also, the request is tagged (not shown in the Figure) using the requester thread id and the corresponding reply will carry the same tag.

When a response from an offload thread is received, the response tag is used to find the corresponding requester thread. When the last response from the offload component(s) of a given processing step is received, the corresponding thread is woken up and moved from the postOff thread list to the preOff thread list to perform the next processing step. Figure 4.10 shows the wake-up logic of a thread.

4.4.3 Pipelined engines

Whenever the control flow between processing steps is solely jumps from one step to its next step, the engine compiler can generate a pipelined engine.

In addition to hiding the latencies of the offload calls, pipelining improves the throughput of an engine and can be useful if the engine needs higher

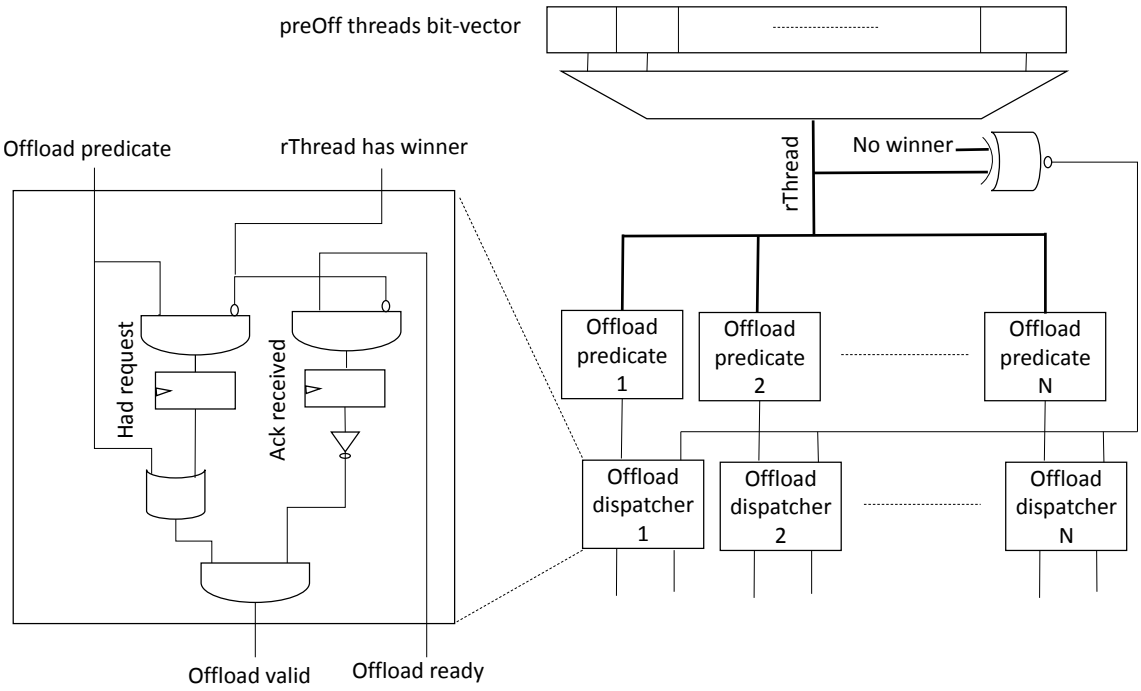


Figure 4.9: Offload dispatch logic.

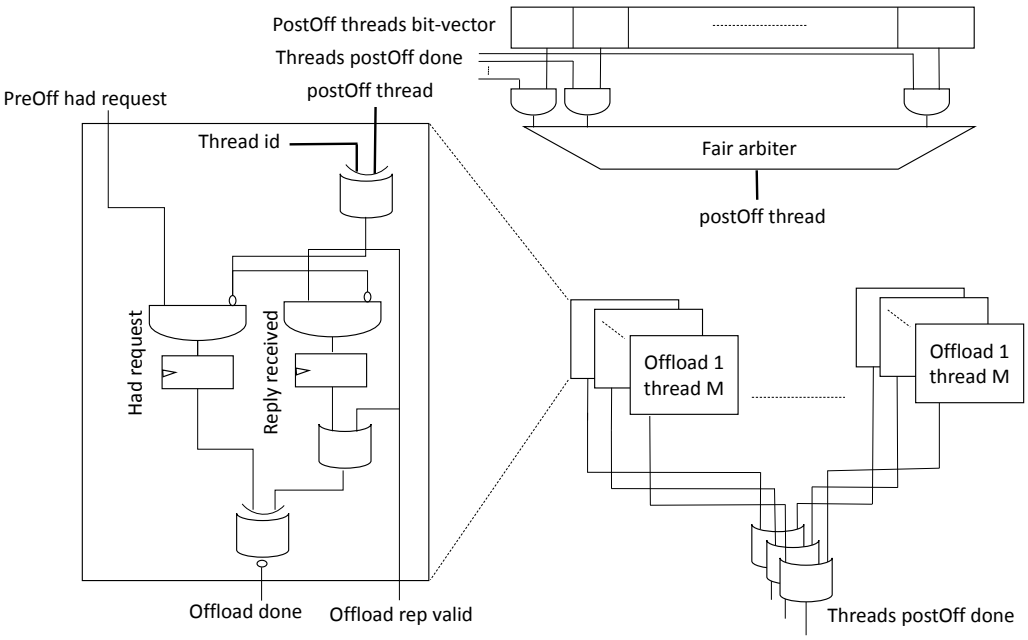


Figure 4.10: Offload thread wakeup logic.

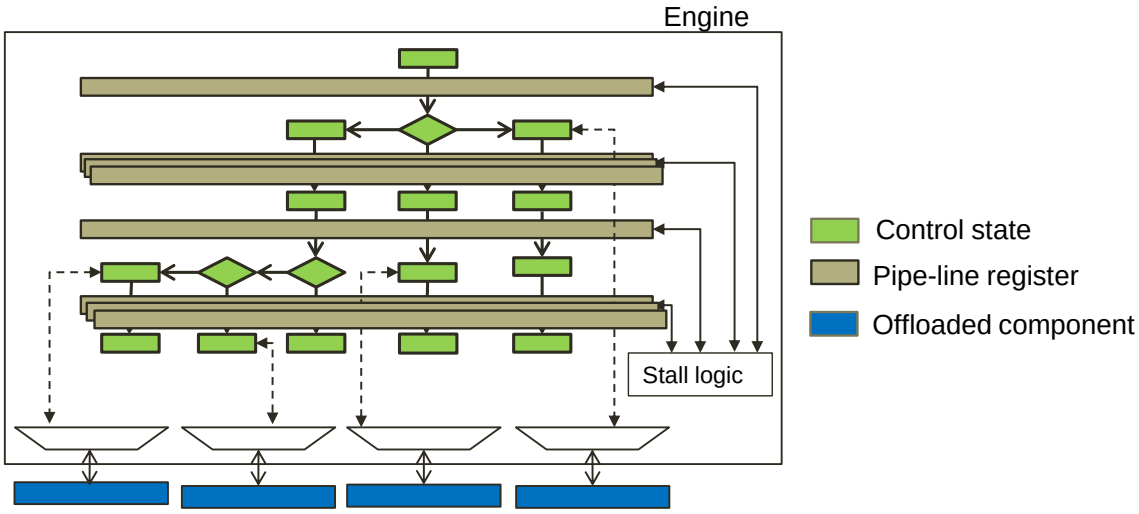


Figure 4.11: A pipeline engine for a loop-free input engine code.

computation throughput.

4.4.3.1 Receive/send stages

A dedicated receive stage is added at the start of the engine's pipeline and a dedicated send stage is added at the end of the engine's pipeline.

4.4.3.2 Split-phase stages

The engine compiler generates a preOff pipeline stage and a postOff pipeline stage for every processing step with a call to an offloaded component.

The structure of preOff dispatch logic and postOff wakeup logic in a pipelined engine is very similar to the corresponding stages in a multi-threaded engine. The main difference, however, is that unlike a multi-threaded engine, which uses the same execution pipeline for all processing steps, a pipelined engine has dedicated preOff and postOff stages for each processing step with

offload calls.

4.4.3.3 Stall logic

In a pipelined engine, in addition to engine interfaces, each pipeline stage follows the latency-insensitive protocol [12]. A preOff stage is stalled when there is not any free pipeline register for accepting the result of the stage (see the next subsection). When a preOff stage is stalled, due to the lack of free offload context, the ready signal is de-asserted to ripple back the stall behavior. A postOff stage is stalled, however, whenever its next stage is stalled. The ready signal of the received pipeline stage is connected to the whole engine ready signal, and the valid signal of the send pipeline stage is connected to the whole engine valid signal.

4.4.3.4 Pipeline registers

A pipelined engine does not have any global context memory. The global variables are kept in pipeline registers. For a given input data element, corresponding global variables are marching along the input data element in the pipeline registers through the pipeline stages.

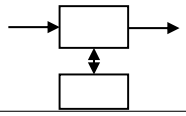
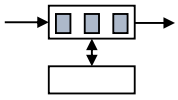
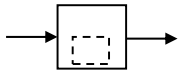
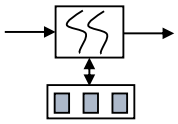
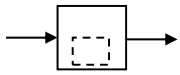
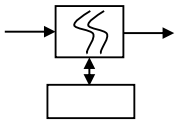
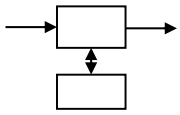
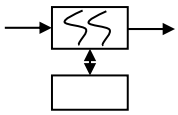
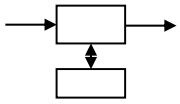
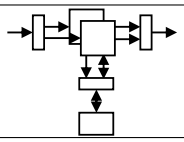
A postOff stage has multiple contexts as its pipeline register, each associated with processing of a different data element waiting for the offload call responses. Similar to a multi-threaded engine, the offload requests in a pipeline engine are tagged using a unique id associated to the context id of the data element. The same tag is used to find the corresponding context when the offload reply is received.

4.5 Refiner

The refiner module in the compiler generates a refined top-level composition code from (i) the original composition code and (ii) a set of refinements. Each refinement consists of a design component that is going to be refined and the corresponding refinement action that is changing the component.

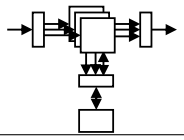
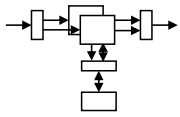
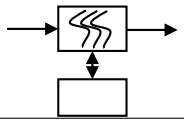
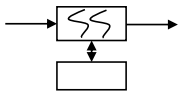
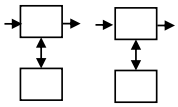
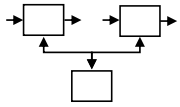
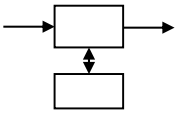
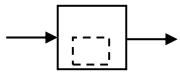
4.5.1 Refinement rules

Table 4.1: Throughput refinements

Refinement	Original pattern	Refined pattern	Criteria	Example:pre-refine	Example:post-refine
Pipelining			Bottleneck, Concurrency safe, Engine, Pipelinable	x = Engine("x.c")	x = PipeEngine("x.c")
Pipe-offloading			Bottleneck, Concurrency safe, Engine with pipelinable onloaded function, High onload rate	x = Engine("xy.c")	x = MTEngine("x.c", 2) y = PipeEngine("y.c") xy = Offload(x,y)
Offloading			Bottleneck, Concurrency safe, Engine with onloaded function, High onload rate	xy = Engine("xy.c")	x = MTEngine("x.c", 2) y = Engine("y.c") xy = Offload(x, y)
Multi-threading			Bottleneck, Concurrency safe, Engine, High offload rate	x = Engine("x.c")	x = MTEngine("x.c", 2)
Replication			Bottleneck, Concurrency safe	x = Engine("y.c")	y = Engine("x.c") x = Replicate(y, 2)

Tables 4.1 and 4.2 list the Gorilla++ throughput and area refinement rules. The table entries contain the criteria that are necessary to activate a

Table 4.2: Area refinements

Refinement	Original pattern	Refined pattern	Criteria	Example:pre-refine	Example:post-refine
Component removal			Under-utilized, replicated component	x = Replicate(y, 3)	x = Replicate(y, 2)
Thread removal			Under-utilized, multi-threaded engine	x = MTEngine("x.c", 3)	x = MTEngine("x.c", 2)
Offload-sharing			Multiple under-utilized, stateless, and offloaded component	u = Offload(x, z) v = Offload(y, z) w = Chain(u, v)	u = Chain(x, y) w = Offload(u, z)
Onloading			Under-utilized, offloading and offloaded engines	x = MTEngine("x.c", 2) y = Engine("y.c") u = Offload(x, y)	u = Engine("xy.c")

rule. Chapter 5 elaborates on the rules criteria. The tables also present an example of composition code before and after applying the rule.

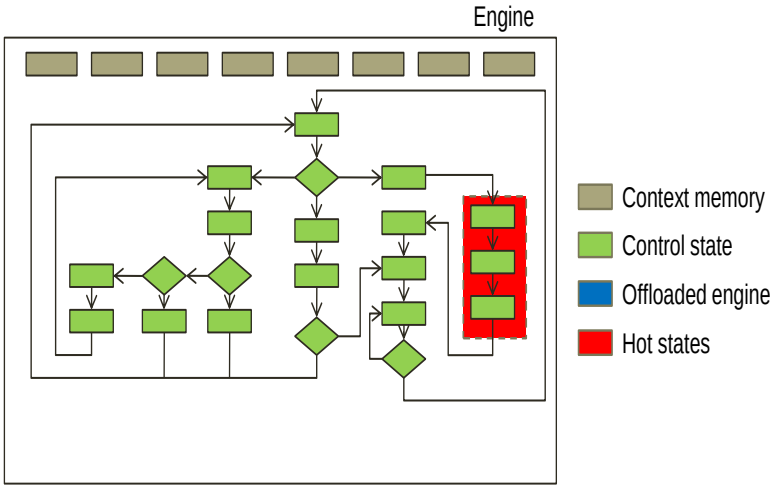
Throughput refinements include pipelining, pipe-offloading, multi-threading, offloading, and replication. Pipelining changes an engine to a split-phase pipeline. Multi-threading transforms a single-threaded engine into a multi-threaded engine or increases the number of threads in an already multi-threaded engine. Unlike pipelining, multi-threading can be applied to any arbitrary engine, even one with a loop in the control flow graph. Offloading combines (i) creating a new stand-alone engine as an offloaded component from an onloaded function and (ii) if necessary, increasing the number of threads in the offloading engine.

Pipe-offloading combines (i) creating a new stand-alone engine as an

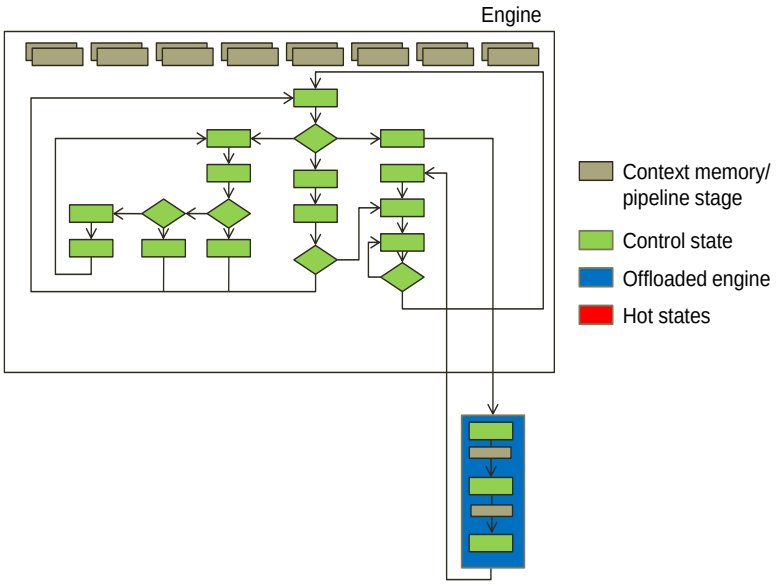
offloaded component from an onloaded function, (ii) pipelining the offloaded engine, and (iii) if necessary, increasing the number of threads in the offloading engine. This rule enables pipelining of the part of the engine that is not efficiently pipelinable as a whole. This is done by separating it into an offloaded engine that can then be pipelined using Gorilla++ split-phase pipeline template (Figure 4.12). Pipe-offloading is the combination of pipelining, offloading, and multi-threading refinements.

The above three refinements can be applied only on engines, and not on composite components. The replication refinement, however, can be applied on both engines and composite components.

The component removal rule can be used to save the area by decrementing the replication factor of under-utilized, replicated components. Similarly, if a thread in a multi-threaded engine is under-utilized, the thread removal rule can be used to save the area by decrementing the number of threads. Offload-sharing (Figure 4.13) is applied when more than one instance of a stateless and offloaded component is under-utilized. In such a case, the offloaded component is shared between offloading ones to save area. The onloading rule, which is the opposite of the offloading rule in the throughput refinements, transforms an offloaded engine into a function in the offloading engine. This is done when there is no need for offloading parallelism. Onloading saves area by eliminating extra overhead associated with two independent engines, including the offload interface area overhead. De-pipelining transforms a pipelined engine to a pipelined one when the extra area overhead associated with the pipelining is not beneficial.

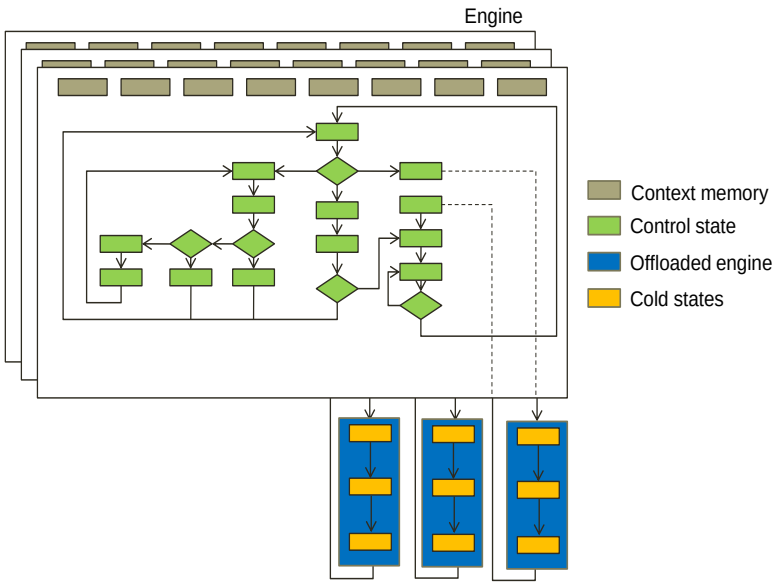


(a) Pipe-offloading refinement - before

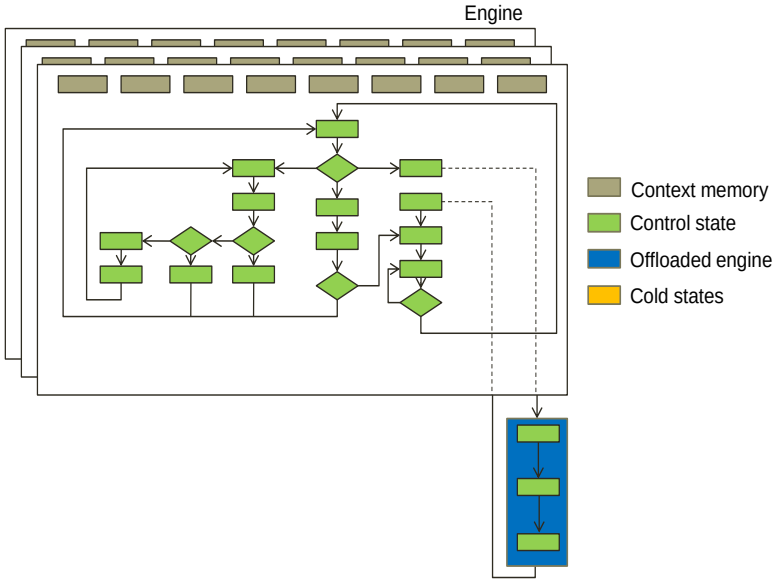


(b) pipe-offloading refinement - after

Figure 4.12: Pipe-offloading refinement to pipeline part of an engine that is not efficiently pipelinable as a whole.



(a) Offload-sharing refinement - before.



(b) Offload-sharing refinement - after.

Figure 4.13: Offload-sharing refinement to share under-utilized, offloaded components with the same type.

4.5.2 Single function assignment (SFA) form and source rewriting

The refiner program rewrites the design composition code in order to refine its micro-architecture. This requires a clear, one-to-one mapping between the design components and their representations in the composition code. Therefore, Gorilla++ creates a SFA form for a composition code and uses this form in the rest of the compilation process. In SFA form, every component is created in a separate assignment. Below is a part of the composition code that creates the **histograms** component in the histogram accelerator, described in Section 3.5 of Chapter 3.

```
val histograms = Replicate(Offload(histogramEngine, div), 2)
```

The equivalent SFA form for the above code contains two statements as follows. Using the SFA form, the composite components, which are the result of **Offload** and **Replicate** compositions, are generated in separate statements.

```
val x1 = Offload(histogramEngine, div)\\
val histograms = Replicate(x1, 2)
```

4.6 Comparing Gorilla++ to closely related work

Gorill [51], the predecessor of Gorilla++, used multi-threading with the support of out-of-order processing of input data elements when offload accesses

have variable latencies. Similar technique were used by Halstead and Najjar and Tan et al. [32, 84] to enable out-of-order execution of data elements in a pipeline.

The technique proposed by Liu et al. [57] is similar to Gorilla++ Chain parallelism. However, it captures the parallelism from the sequential description and therefore it can be used as a complement to Gorilla++.

Gorilla++ refinement technique based on rewriting the functional description is similar to SAFL+. However, unlike SAFL, Gorilla++ uses pure sequential notion for designing the leaf components and only use functional composition at the coarse granularity. Also, the micro-architecture refinements in Gorilla++ have much more variety. Finally, Gorilla++ uses a systematic method, based on detecting inefficiencies in the design, to choose a particular refinement among all refinements.

Gorillla++ refinements that are related to the offload connections including multi-threading, offloading, pipe-offloading, and offload-sharing are not part of SDF-based refinements, since offloading is not a first-order connection type in SDF-based HLS.

Chapter 5

Gorilla++ rule-based design space exploration

5.1 Related work: design space exploration in HLS

In order to find the right micro-architecture to capture the coarse-grain parallelism in the application, an HLS tool should (i) build the design space of different micro-architecture alternatives and (ii) explore the design space to find the most efficient alternative in terms of performance, area, and/or power consumption. This section describes the related work to the Gorilla++ rule-based DSE with respect to these two subtasks.

5.1.1 Building the design space

5.1.1.1 Parametric designs

One solution to build the micro-architecture design space is to rely on the programmer to encode the micro-architecture design space explicitly. Building a highly parametric design is more complex than a non-parametric one. However, in many cases, the benefit of using a parameterized design, specially when some of the design constraints are not predictable at the early stage of the design process, outweighs the initial design cost [78]. A class of hardware description languages, including Genesis [79] and Chisel [4], facilitate this design approach with special language constructs for parameterization.

5.1.1.2 Refinement patterns

An alternative method to build the design space is to automatically perform refinements in the design using a compiler. Two examples of these methods are presented in Section 4.1 of Chapter 4 as (i) refinement of rule-based designs and (ii) refinement of SDF-based designs.

5.1.2 Exploring the design space

5.1.2.1 Closed-form formulations

StreamIt showed that by using the notion of balance equations in SDF and little's law along the work estimates of dataflow nodes, the problem of selecting the types and degrees of parallelism can be solved using closed-form formulations.

Cong et al. [20] used a similar approach for finding (i) the degree of pipelining and (ii) the replication factor of each node using static formulation. They used the area of pre-compiled individual modules to estimate the total accelerator area. Li et al. [54] presented a formulation for finding the right set of merging and splitting refinements for dataflow nodes. They used an area model consisting of the interface area and the main logic area for each dataflow node to estimate the refined node areas after merging or splitting.

5.1.2.2 Generic optimization methods

Generic optimization algorithms can be used to explore the design space. Given a utility function that specifies the quality of results for a given performance and area and based on the generated design space for a design, these algorithms can find an optimal point in the design space to find an optimal utility function. Hill climbing [35] and simulated annealing [82] are two

prominent examples of such generic optimization algorithms.

5.2 Performance counters

Table 5.1: Major Gorilla++ performance counters

Performance counter	Details
Back-pressure	# of cycles in which a source component has a data element to transfer but the sink component cannot accept the data
Under-utilization	# of cycles in which a sink component is ready to accept a data element but the source component does not have any data to send
In/out throughput	# of received or sent data elements/total cycles
Offload rate	# of cycles waiting for an offload/number of execution cycles
Processing step utilization	# of cycles an engine spends on each processing step/number of execution cycles

Gorilla++ components have a set of performance counters (Table 5.1) that can be used to find and eliminate possible inefficiencies in the design. Performance counters can be implemented as part of the accelerator hardware. A dedicated ring network is automatically inserted in the design to access the counters. During the auto-refinement process, Gorilla++ extracts the counters periodically and reports them to the refiner program. Performance counters are implemented within components' abstract classes and, therefore, are completely transparent to the Gorilla++ programmers.

Figure 5.1 shows the overhead of performance counters on the area

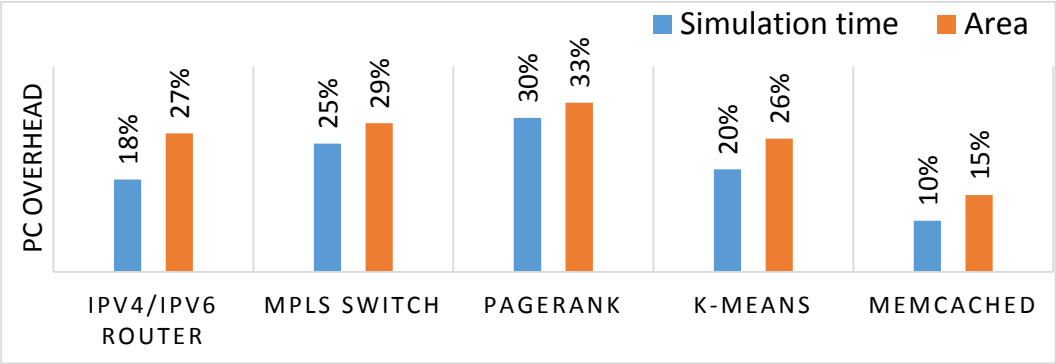


Figure 5.1: The overhead of performance counters on the area and simulation time of different benchmarks.

and simulation time of different Gorilla++-generated accelerators, described in Chapter 6. Performance counters and their associated logic have an average simulation time overhead of 20.6% and an average area overhead of 26%. After the auto-refinement process, the design can be recompiled without performance counters, eliminating the area overhead. Since the performance counters are non-intrusive, they do not have any overhead on the throughput of accelerators in terms of generated data elements per cycle. The performance counters, however, may affect the maximum clock frequency of the accelerators during the profiling phase.

5.3 Rule activation criteria

This section elaborates the criteria of activating the refinement rules in Tables 4.1 and 4.2.

5.3.1 Bottlenecks

Throughput refinements are applied only on bottleneck components. A bottleneck component is one whose input is back-pressured (Table 5.1) but neither its output nor its offload interfaces are back-pressured (i.e., $inputBackPressure > \alpha$[1] and $outputBackPressure \leq \alpha$ and $\forall i,\ 1 \leq i \leq numberOfOffloads$, $offloadBackPressure(i) \leq \alpha$). If either one of the offload ports or the output port has a high back-pressure rate, the component is not considered a bottleneck. In this case, there is a downstream component in the design that is the bottleneck. We assume that there is no back pressure on the primary output of the design.

5.3.2 Engines with high offload/onload rate

Multi-threading (Table 4.1) refinement is applied only if the engine's offload time (Table 5.1) is large enough to overlap the compute time of an extra thread (i.e. $offloadRate \geq (1/\beta) * numOfThreads * (1 - offloadRate)$).[2] Figure 5.2 shows an example of an engine with threads that have large enough offload times. In this case, the refinement changes the number of threads to four (from three). This is because the sum of three compute times is less than a single offload time. Therefore, ideally, at any given time, an engine can have one thread to perform computation while three other threads waiting for the offload. The same condition is used for the offloading and pipe-offloading refinements to check whether increasing the number of threads in the offloading engine is beneficial. The offload time is estimated as the original engine's

[1] α $(0 < \alpha < 1)$ is a constant number

[2] β constant $(0 \leq \beta \leq 1)$ is introduced to avoid thrashing when the conditions are satisfied only marginally.

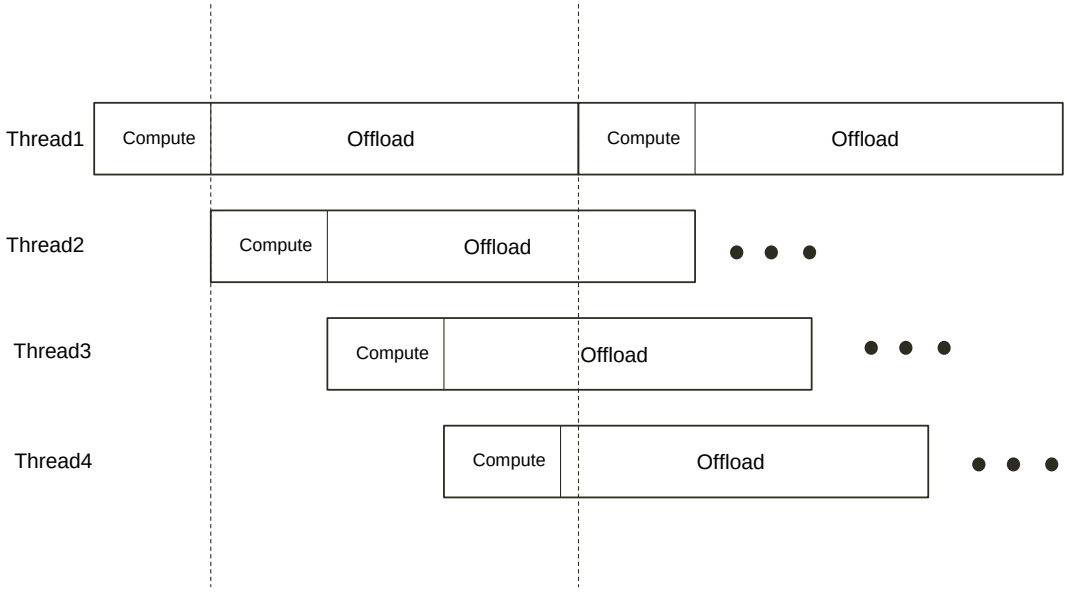


Figure 5.2: Calculating the number of threads based on the thread's compute and offload time.

offload time plus the compute time of the onloaded function. Also, the compute time is estimated as the original engine's compute time minus onloaded function compute time.

5.3.3 Under-utilization

In a replicated component, if the amount of under-utilization is bigger than $1/(\beta * replicationFactor)$,[2] the component removal rule decrements the replication factor to save the area. Similarly, in a multi-threaded engine, if the amount of under-utilization is bigger than $1/(\beta * numOfThreads)$,[2] the number of threads is decremented. When there is an offload relationship between two engines and both of the engines have an under-utilization more than γ , the onloading refinement merges two engines into a single engine.

In offload sharing refinement, the utilization of the new, shared component is projected from the utilization of non-shared ones based on the $1/\beta * \sum_{c \in C}(1 - underUtilization(c))$ formula[2], with C denoting the set of non-shared instances of the same component. The refinement is not applied if the new utilization is greater than one.

5.3.4 Safety of refinements

Throughput refinements are only allowed for concurrency-safe components. An engine is concurrency-safe if either (i) it does not have any offloaded memory components, either directly or indirectly through intermediate offloaded components or (ii) it is annotated by the programmer as a concurrency-safe engine. In the latter case, the programmer must introduce the necessary locks to ensure data-race-freedom for accesses to the shared, offloaded memory components.

A composite component is not concurrency-safe, if it includes a memory or a lock component or if it includes a non-concurrency-safe component. All other composite components are concurrency-safe.

5.4 Refinement algorithm

The goal of the refinement algorithm is to explore the design space and maximize a utility function for a given (i) injected throughput and (ii) available area resources (areaCeiling).

Given a (i) throughput for the refined design (denoted as Tr) and a throughput for the baseline design (denoted as Tb), (ii) FPGA lookup table utilization for the refined design (denoted as Lr) and FPGA lookup table

utilization for the baseline design (denoted as Lb), and (iii) FPGA DSP utilization for the refined design (denoted as Dr) and FPGA DSP utilization for the baseline design (denoted as Db), a utility function is defined using the following formula.

$$Utility = \begin{cases} (\theta * Tr/Tb) - (\eta * (Lr/Lb + Dr/Db)) & area \leq areaCeiling \\ 0 & area \geq areaCeiling \end{cases}$$

The utility function scalarizes the throughput and area into a single number.[3] The number is used to determine whether the new design is closer to the performance goals. For the benchmarks that do not use any DSP blocks, the corresponding term is omitted.

The algorithm (Figure 5.3) iteratively optimizes the design by applying the refinement rules in two phases. In the first phase, only throughput refinements are applied and in the second phase only area refinements are applied. The algorithm ignores the refined design if the design utility is not improved.

In each phase, the algorithm goes through the design components and for each component determines which Rex refinement rules are applicable to the component. The rules are searched based on a static priority that corresponds to their order in Tables 4.1 and 4.2. For the throughput optimization phase, the algorithm considers the lower-level components in the design hierarchy before the higher-level ones. For the area optimization phase, the algorithm considers the higher-level components before the lower-level ones. This is to minimize the area overhead of throughput refinements and maximize the area saving of area refinements. The rule criteria are checked before

[3] θ and η constants are parameters of the refinement algorithm, which can be set by the user.

applying the refinement on a particular component. If the utility is not improved, the algorithm proceeds to the next rule or the next component. When all the components are exercised for the throughput optimization phase, the algorithm switches to the area optimization phase. The algorithm terminates when all components are visited in both phases.

Figure 5.3 shows the pseudo-code for the Gorilla++ refinement algorithm. The auto-refinement process optimizes for a single objective, either throughput or area, at any iteration. The algorithm determines which Gorilla++ refinement rules are applicable to each individual component in the design. For throughput optimization, the algorithm considers the lower-level components in the design hierarchy before the higher-level ones. For area optimizations, the algorithm considers the higher-level components before the lower-level ones. This is to minimize the area overhead of throughput refinements and to maximize the area saving of area refinements. The rule criteria (Tables 4.1 and 4.2) are checked before applying the refinement on a particular component.

If the utility is not improved by the refinement, the algorithm backtracks and the new micro-architecture is ignored. In such a case, the algorithm proceeds to the next rule or the next component. When all the components are exercised for throughput refinements, the algorithm switches to the area mode.

```
refine() {
 /*Variable declarations not shown*/
 utility = baseMicroArchUtility();
 components = parseDesign(); components.sort(bottumUp);
 while (!refineStep(throughputObjective, components, utility));
 components = parseDesign(); components.sort(TopDown);
 while (!refineStep(areaObjective, components, utility));
}

boolean refineStep(objective_t objective,
 components_t components, double &lastUtility) {
 /*Variable declarations not shown*/
 for (c = components.begin(); c != components.end(); c++) {
  for (r = rules[objective].begin();
   r != rules[objective].end(); r++) {
   if (r.criteria(c, objective)) {
    r.test(c, &throughput, &area); //Synthesis and simulate
    newutility = utilityFunc(throughput, area);
    if (newUtility > utility && area < areaCeiling &&
     throughput > throughputFloor) {
     r.applyRefinement(c); //Change the design
     utility = newUtility; return false;}}}}
 return true;}
```

Figure 5.3: Pseudo-code of the Gorilla++ refinement algorithm.

5.5 Refining the case study histogram accelerator

5.5.1 Rule-based refinement

Figure 5.4 shows the refinement steps for the histogram accelerator and the effect of each refinement on the throughput, area, and back-pressure values of the histogram and cache controller engines. The first refinement is pipe-offloading the "classify" function into a separate pipelined engine shared by histogram engines (Section 4.5.1).[4] The rest of the refinements increase the number of threads in the histogram and cache controller engines. Back-pressures show how bottlenecks are moved between the two engines during the refinement steps. Since the tester is always ready to inject a new value, the final back-pressure on the histogram engine is still high. However, the refinement process stops due to the diminishing return of increasing the number of threads.

The same experiment is repeated for three different histogram input data sets. For the first data set, dense histogram (DH), the tester sends random input numbers in the range of the first 16 buckets. Therefore, most of the accesses to the cache are hit accesses (cache size is 1,024 words). For the second experiment, sparse histogram (SH), the tester sends random input numbers in the range of all 8,192 buckets. In this experiment, however, most of the accesses to the cache are miss accesses. For the third experiment, the tester mixes the first and the second input data sets with equal probability. For each of these three input data sets, Gorilla++ generates a different micro-architecture.

A small micro-architecture is generated for the DH input with two histogram engine threads and two cache controller threads. Since this input

[4]Note that due to a control flow loop in the report code, the histogram engine cannot be pipelined by itself.

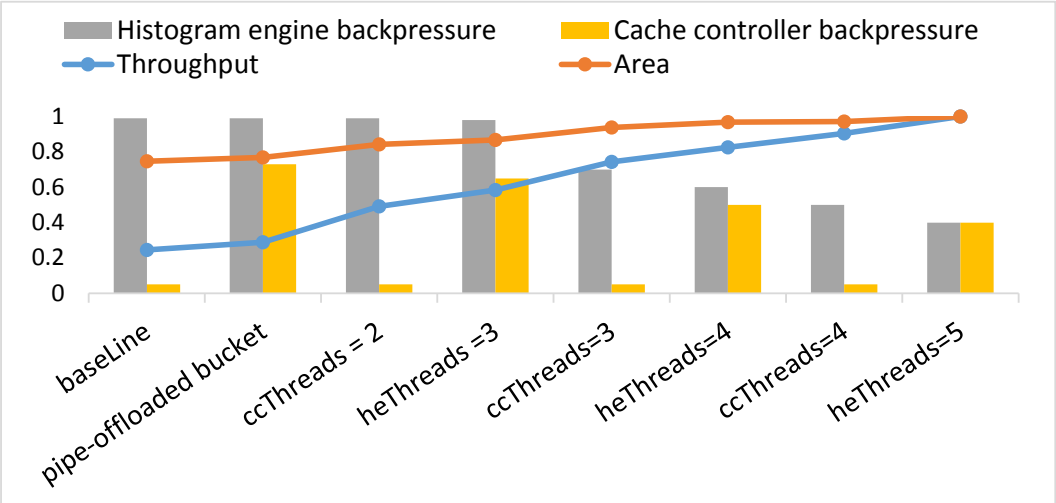


Figure 5.4: The refinement steps of the histogram accelerator. Number of buckets = 8,192. Throughput and area numbers are normalized to their maximum values.

data set causes fewer cache misses, a micro-architecture with less memory-level parallelism, is generated. A medium micro-architecture is generated for the mixed input data set with three histogram engine threads and three cache controller threads. A large micro-architecture is generated for the SH input, with four histogram engine threads and five cache controller threads. This micro-architecture has a high enough memory-level parallelism to tolerate the high miss-rate.

Figure 5.5 shows how the throughput is changed when each of these three input data sets is used on each of the three generated micro-architectures. For a given input data set, using a different micro-architecture than the one customized for the data set results either in a lower throughput or a higher area.

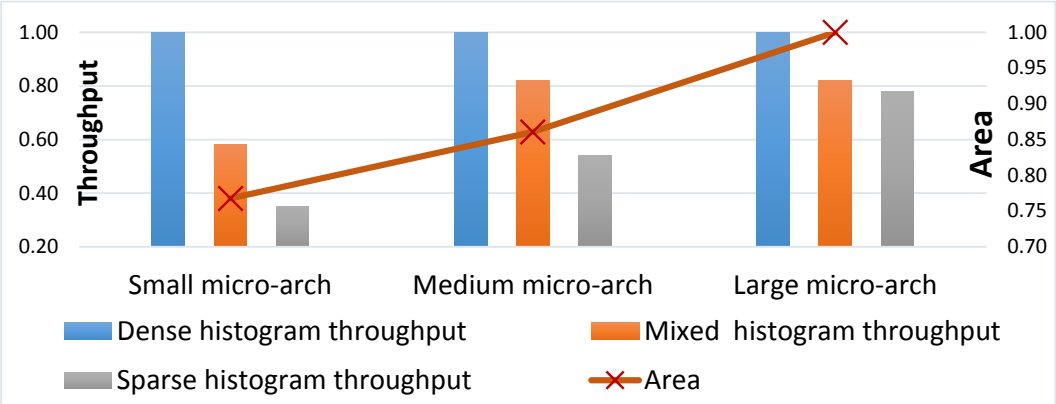


Figure 5.5: Throughput of three different input data sets on three Gorilla++-generated micro-architectures for the histogram accelerator. Throughputs are normalized to the throughput of dense histogram input data. Areas are normalized to the area of large micro-architecture.

5.5.2 Manual construction of the design space

In order to compare a Gorilla++ rule-based DSE with a generic optimization algorithm, a parameterized version of the design is required. Gorilla++, however, does not require a designer to parameterize the design explicitly. Figure 5.6 shows a parameterized version of the histogram accelerator. The **offloadedClassifier** parameter specifies whether the classifier function is an onloaded function, an offloaded pipelined engine, or an offloaded multi-threaded engine. The **numOfHistogramThreads** parameters specifies the number of threads in the **histogramEngine**. The **numOfClassifierThreads** determines the number of threads in the **classifer**, if it is implemented as a stand-alone offloaded engine. The **numOfCacheThreads** determines the degree of memory-level parallelism in the blocking cache. Finally, the **numOfHistogramEngines** specifies the replication factor that is used for replicating the **histogramEngines**.

```
class Top extends gComponent with include {
 val inputGenerator = Engine("inputGenerator.c")

 val histogramEngine = if (__offloadedClassifier__ == ONLOADED) {
  yield MTEngine("histogramEngine.c",
   __numOfHistogramThreads__)
 } else if (__offloadedClassifier__ == PIPEOFFLOADED) {
  yield Offloaded(MTEngine("histogramEngine.c",
   __numOfHistogramThreads__),
   PipeEngine("classifier.c"), classifier)
 } else if (__offloadedClassifier__ == OFFLOADED) {
  yield Offloaded(MTEngine("histogramEngine.c",
   __numOfHistogramThreads__),
   MTEngine("classifier.c", __numOfClassifierThreads__),
    "classifier")
 }
 val outputReporter = Engine("outputReporter.c")
 val hCache = Cache(height=1024, lineSize=128, tagSize=20,
  numOfThreads = __numOfCacheThreads__)
 val memory = Offlod(hCache, DRAM)
 val histogramLock = lock(BUCKETS)
 val div = FPDivider()
 val histograms = Replicate(Offload(histogramEngine, div),
  __numOfHistogramEngines__)
 val histogramsAndIO= Chain(inputGenerator, histograms,
  outputReporter)
 val result = Offload(histogramAndIO, memory, histogramLock)
 }
```

Figure 5.6: Parameterized histogram composition code.

Chapter 6

Gorilla++ in Practice

6.1 In-line acceleration

It is important to understand the role of Gorilla++-generated accelerators in the whole system. One interesting possibility is to use them as in-line accelerators. In-line accelerators [50] are a broad category of accelerators for NBD applications that process incoming packets right after they are received from the network.

As shown in Figure 6.1, an in-line accelerator sits between the network interface and the general-purpose cores and intercepts incoming packets coming from the network. In its general form, an accelerator can (i) process the packets completely without involving general-purpose cores , (ii) process the packets partially, leaving the rest of the computation for the general-purpose cores, or (iii) pass through the packets to the general-purpose cores without processing them.

The important characteristic of the in-line acceleration notion is that applications can be sliced into (i) a simple, fast-path that is executed by the accelerator and (ii) a complex, slow-path that is executed by the general-purpose cores. This notion decreases the complexity of the accelerator and leverages the "optimize the common case" design principle. The expensive hardware resources are spent on the part of the application that is frequently used and consequently essential to increase the overall application performance.

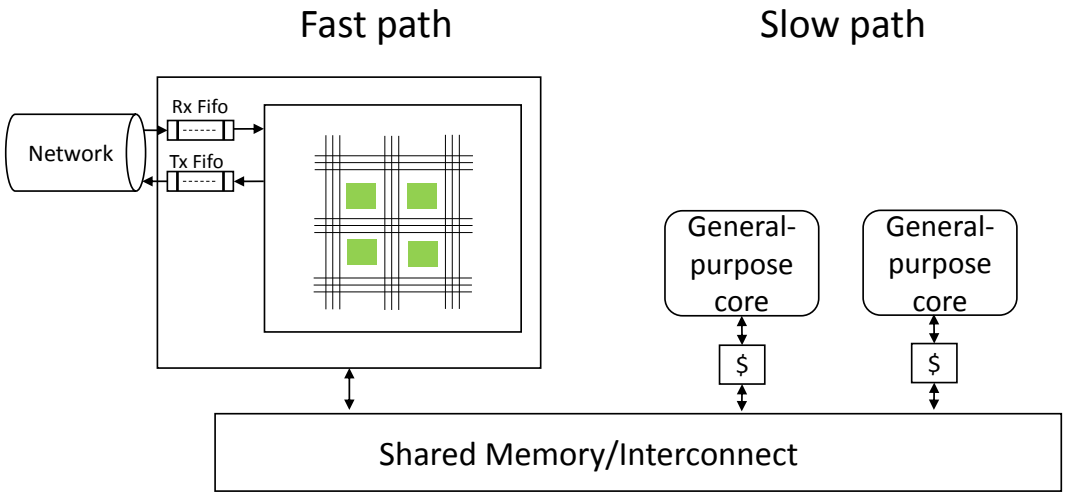


Figure 6.1: The role of in-line accelerators in a system.

Although this technique was previously used such as in high-end packet processing systems by manually splitting the fast-path (data plane) and slow-path (control plane), we proposed a semi-automated technique [50] for slicing a server application into an accelerated fast-path and a slow-path running on a general-purpose processor.

Handling a high throughput of IO packets, can cause inefficiency in traditional general-purpose processors. Several techniques have been proposed to resolve this problem, including integrated NIC [5], cache intervention on NIC [41], and user-level networking protocol with direct access to low-level packet buffers [65]. In-line acceleration eliminates the copying of the packets completely by moving the computation into the NIC rather than moving the data from NIC to the main memory.

For all the benchmark applications presented in this chapter, the Gorilla++-

generated hardware can be used in a system as an in-line accelerator.[1]

Table 6.1 shows the details of the benchmark applications, including their sources of irregularity and the number of lines of code (LoC) for (i) the Gorilla++ description and (ii) the generated Chisel description. The table also includes the injected input throughput to the accelerator under the test[2] .

6.2 Experimental setup

The Gorilla++-generated Chisel is compiled to Verilog, which is later synthesised and placed and routed using the Xilinx ISE 13.1 tool set on a Virtex-6HX255T device. In these experiments, the areaCeiling is set to 30% of the FPGA resources (Figure 5.3). The α, β, and γ constants (Section 5.3) are set to 20%, 80%, and 50% respectively. The constants in the utility function (Section 5.4), θ and η, are set to 150,000 and 10,000, respectively. The above parameters can be changed by the Gorilla++ user to set different trade-offs between exploration time, area, and throughput.

6.2.1 Characteristics of the baseline generic optimization algorithms

We compare the Gorilla++ refinement algorithm against two generic optimization algorithms, hill climbing [35] and simulated annealing [48]. During the optimization process in the design space, the hill climbing switches to

[1]The K-means and PageRank accelerators are setup as a single stand-alone hardware in order to make their evaluation simpler. However, since their streaming architecture is adopted from Dryad [42] project, both of these accelerators can be changed to a multi-computer setup.

[2]The primary reason to use a throughput less than a one is to avoid long exploration time by generic optimization algorithms

the first neighbor design point that has a higher utility. Simulated annealing, however, may accept design points with a lower utility than the current utility during the optimization process. This enables the optimization to escape from local maximums in many cases. Accepting a lower utility is controlled by a probabilistic parameter, **temperature** parameter. A higher temperature increases the probability of accepting a lower utility design point. On the other hand, a lower temperature decreases the probability of accepting a lower utility design point. The algorithm starts with a high temperature value and decreases the temperature gradually. A zero temperature causes the algorithm to behave like a hill climbing.

In our experiments, the simulated annealing starts with a temperature value of 50000 and then decreases the temperature in five rounds. In each round, the number of iterations is multiplied by a factor of 1.5x. The initial round has 10 iterations. There is no limit on the number of iterations for the last round which has a zero temperature. A design with a lower utility is accepted if the result of $e^{(newUtility-oldUtility)/temprature}$ is greater than a random number between zero and one.

Unlike Gorilla++, in order to use these generic optimization algorithms, a parameterized version of the design is required. We make sure that the design parameters encode the same design space that Gorilla++ explores.

6.2.2 DRAM model

For PageRank and Memcached benchmarks, which use a DRAM as part of the design, a DRAM model is written using Gorilla++. The DRAM model consists of 1 Rank * 8 Chips * 8 Banks * 8 DRAM arrays * 16Mbits. Row buffer size is 4KBytes resulting in 12bits for column address and 12bits

for row address. Row buffer hit latency, row buffer conflict latency, and address/command/data transfer latency are modeled as 20ns, 60ns, and 10ns respectively. The model services the DRAM requests based on first ready - first come first server (FR-FCFS) scheduling [73].

Table 6.1: Characteristics of the Gorilla++ benchmark applications

Application	Description	Sources of irregularity	Gorilla++ LoC (Top-level / engines)	Generated Chisel LoC	Injected throughput (data elements/cycle)
IPv4/IPv6 router	Layer three IPv4/IPv6 router with direct lookup and QoS counting [51]	Different protocols, accesses to shared QDR interfaces	28 / 280	1,385	1
MPLS switch	Layer two-and-half MPLS switch with two-level tag lookup and QoS counting [51]	Different number of MPLS tags, accesses to shared lookup and QDR interfaces	34 / 312	1,643	0.5
K-means	Iterative, parallel K-means clustering algorithm [42]	Access to shared distance calculators	28 / 463	1,532	0.005
PageRank	Iterative, parallel rank calculation algorithm for web pages [42]	Different number of links per page, access to cache/DRAM, access to shared rank calculators/rank accumulators	33 / 491	1,931	1
Memcached	Fast path get-only handler for Memcached key-value server [50]	Pointer chasing in hash table, access to cache/DRAM, access to shared hash engines	22 / 820	2,910	1

6.3 IPv4/IPv6 header processor

IP routers are layer 3 networking devices that process incoming packets to determine the output port they should be forwarded. This is done by

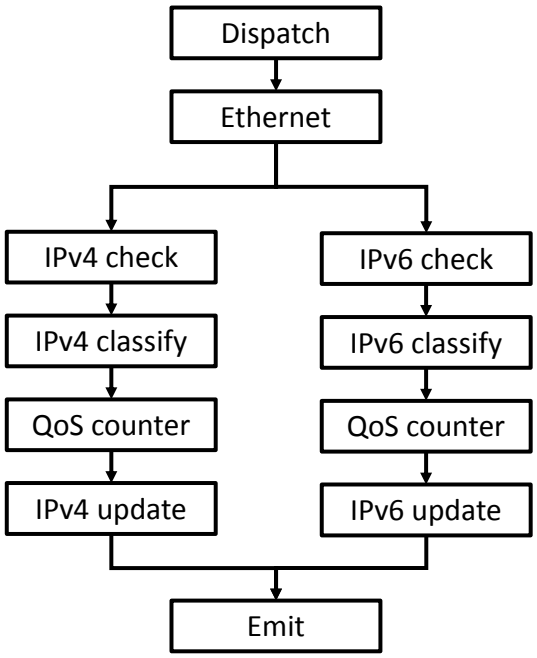


Figure 6.2: Processing steps for the IPv4/IPv6 header processor.

inspecting and updating layer 2 and layer 3 packet headers. Figure 6.2 shows the main processing steps of an IPv4/IPv6 router. The entry point in the processing of a packet is the *Dispatch* step. It checks the layer 2 protocol and jumps to the appropriate state for processing the protocol header. In Figure 6.2 only one layer 2 protocol, Ethernet, is shown. The *Ethernet* step detects the layer 3 protocol and also checks the integrity of the Ethernet header.

Processing the layer 3 protocol, either IPv4 or IPv6, consists of extracting the packet fields, checking the integrity of the fields, classifying the packet by looking up the source and destination addresses, incrementing the quality of service (QoS) counters, and finally updating the packet header fields. The destination port for the packet is determined using a lookup process and the

packet is forwarded to that port. Processing IPv6 is similar to IPv4, except that IPv6 has a different header format with longer source and destination addresses. The exit point for processing a packet in Figure 6.2 is the *Emit* step.

Like our previous work [51], a trie-based IP lookup algorithm is used. The IPv4 lookup algorithm divides a 32-bit IPv4 address into three chunks of 20 bits, 4 bits, and 8 bits. At each stage, the memory address is calculated using the current address chunk and the value returned from the memory in the previous stage. If the entry is a leaf entry, no further read requests for that particular lookup are made. The IPv6 lookup accelerator is similar to the IPv4 lookup accelerator, but the number of trie levels is six instead of three. Therefore, each IPv6 lookup accelerator may have up to twice as many memory accesses as IPv4 lookup. The QoS counter requires six accesses to memories, including three read accesses and three write accesses. To measure the performance of the generated IP header processor, we use IPv4 and IPv6 headers randomly with equal probability.

6.3.1 Base micro-architecture

The Gorilla++ composition code of the IPv4/IPv6 header processor is shown in Figure 6.3. The corresponding base micro-architecture is shown in Figure 6.4. An **IP engine** processes the IP headers. The engine consists of IPv4 and IPv6 onloaded lookup functions as well as a QoS counter onloaded function. Four QDR memories keep the lookup and QoS data. In our implementation, we model each QDR memory using three on-chip scratch-pad memories. Due to limited on-chip memory capacity, however, a small lookup table is used. As shown in our previous work [51], a single QDR chip [71]

```
class Top extends gComponent with include {
  val ipEngine =  Engine("ipEngine.c")
  val mems = for (i <- 0 until 8) {
   yield spMem(height=1024, width=32, "MemInitialConent" + i)
  } ++ for (i <- 9 until 11) {
   yield spMemRW(height=1024, width=64, "MemInitialConent" + i)
  }
  val qdr1Mem = mems.slice(0, 2)
  val qdr2Mem = mems.slice(3, 5)
  val qdr3Mem = mems.slice(6, 8)
  val qdr4Mem = mems.slice(9, 11)
  val memOffloads = ArrayBuffer((qdr1Mem(0), "lookupv4Mem1"),
   (qdr1Mem(1), "lookupv4Mem2"), (qdr1Mem(2), "lookupv4Mem3"),
   (qdr2Mem(0), "lookupv6Mem1"), (qdr2Mem(1), "lookupv6Mem2"),
   (qdr2Mem(2), "lookupv6Mem3"), (qdr3Mem(0), "lookupv6Mem4"),
   (qdr3Mem(1), "lookupv6Mem5"), (qdr3Mem(2), "lookupv6Mem6"),
   (qdr4Mem(0), "qosMem1"), (qdr4Mem(1), "qosMem2"),
   (qdr4Mem(2), "qosMem2"))
  val result = Offload(ipv4Engine, memOffloads)
}
```

Figure 6.3: Composition code for an IPv4/IPv6 header processor.

with four million words capacity is big enough to accommodate lookup tables for four RIPE [72] routing tables[3]. Each QDR channel was attached to one such QDR chip. A shim logic was used to interface to QDR memory. The shim provided the exact same input/output and timing as three independent, on-chip memories. Therefore, the same accelerator, when attached to off-chip QDR memories, is capable of processing headers with large lookup tables.

[3](i) rrc00, Ripe NCC Amsterdam, (ii) rrc01, Linux London, (iii) rrc02, Sfinx Paris, and (iv) rrc16, Miami

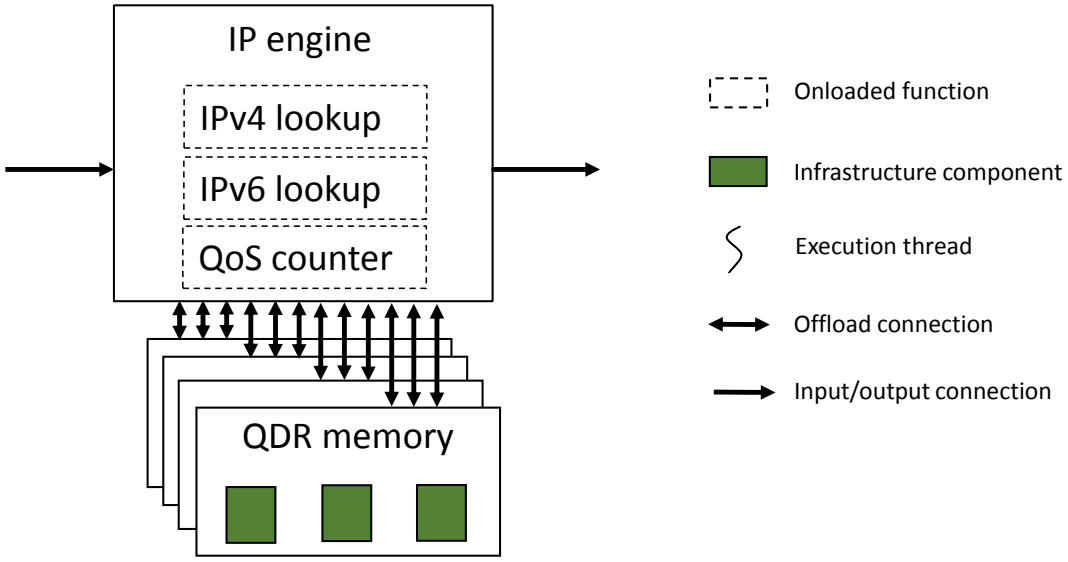


Figure 6.4: IPv4/IPv4 header processor base micro-architecture.

6.3.2 Refined micro-architecture

Gorilla++ refines the micro-architecture of the IPv4/IPv6 header processor in 19 iterations. The refined micro-architecture, shown in Figure 6.5, consists of 12 replicated **IP engine**s. Each engine has three execution threads. Engines are connected to shared and pipelined **IPv4 lookup**, **IPv6 lookup**, and a **QoS counter**.

A parameterized version of the composition code with the parameters specified in Table 6.2 is written and used to generate a refined micro-architecture using hill climbing and simulated annealing DSE algorithms. The result of exploring the design space using generic optimization algorithms as well as the Gorilla++ rule-based DSE is shown in Figure 6.6. The horizontal axis shows the number of iterations for each change in the design. The vertical axis shows the utility function (Section 5.4). The hill climbing algorithm ex-

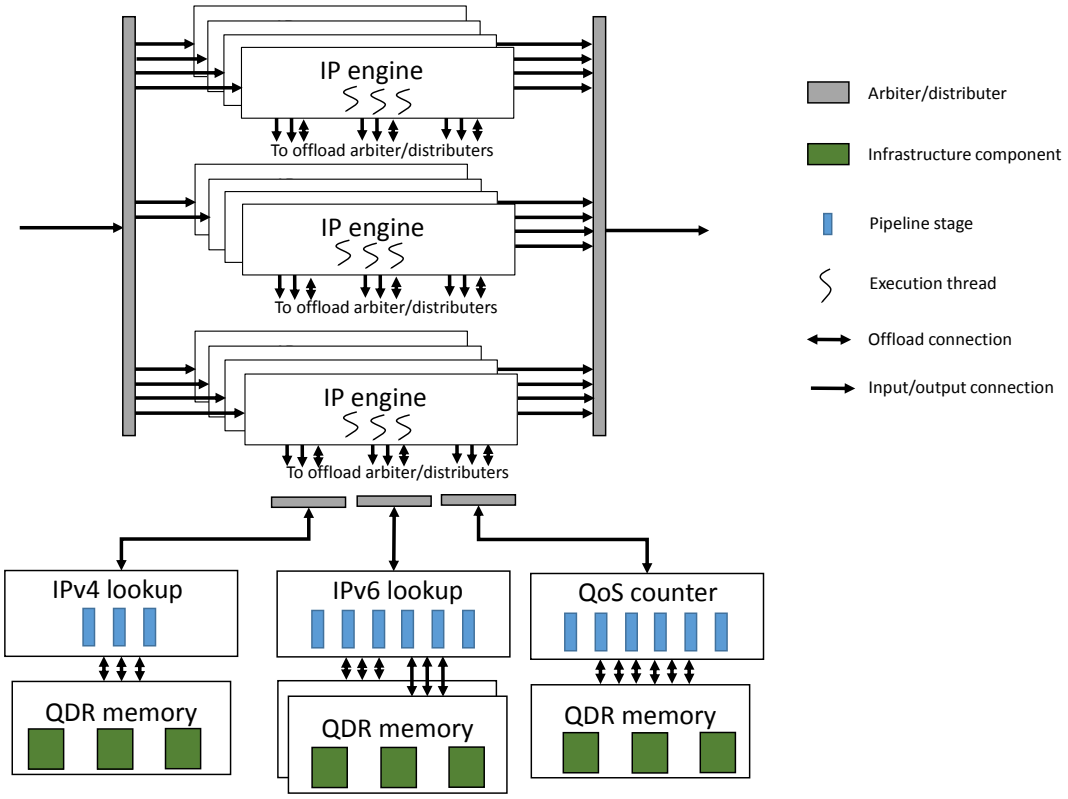


Figure 6.5: IPv4/IPv4 header processor refined micro-architecture.

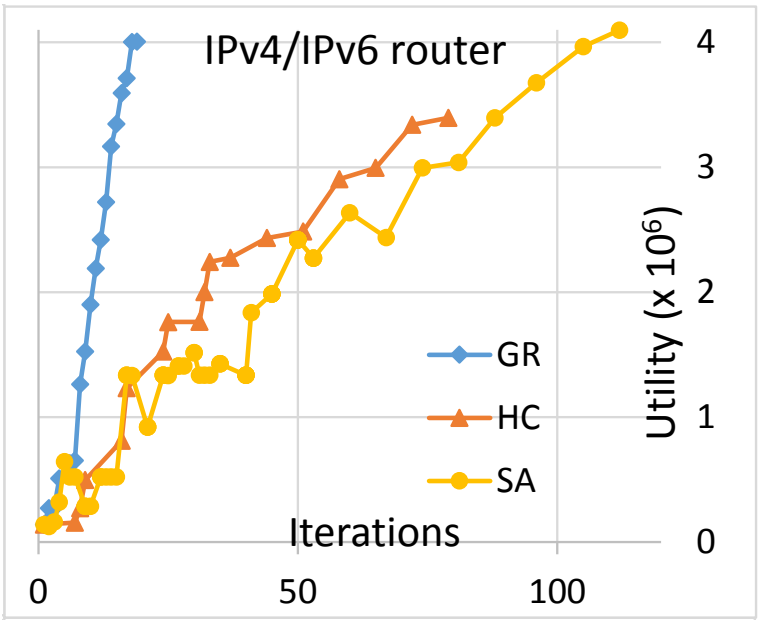


Figure 6.6: DSE of the IPv4/IPv6 header processor (i) HC:hill climbing, (ii) SA:simulated annealing, and (iii) GR:Gorilla++ refinements.

Table 6.2: Parameters in the manually parameterized IPv4/IPv6 header processor

Parameter	Type	Range	Initial value
IPv4 lookup offload type	offload type	onloaded, offloaded, pipeOffloaded	onloaded
IPv6 lookup offload type	offload type	onloaded, offloaded, pipeOffloaded	onloaded
QoS counter offload type	offload type	onloaded, offloaded, pipeOffloaded	onloaded
Number of IP engine threads	numeric	1-8	1
Number of IP engines	numeric	1-16	1

plores the design space in 78 iterations and the simulated annealing algorithm explores the design space in 112 iterations.

Table 6.3 shows the results of the generated IPv4/IPv6 header processors using different DSE methods including the throughput, number of LUT resources, number of register resources, and the maximum clock frequency. Gorilla++ and simulated annealing generate the same micro-architecture. Hill climbing, however, generates a micro-architecture with more threads and fewer engines. This is because hill climbing chooses multi-threading six times, even when it provides a small utility improvement. Simulated annealing, however, can revert this decision and increase the number of engines instead. When refinement-based DSE is used, the offload rate criteria (Section 5.3.2) ensures that Gorilla++ uses multi-threading when it provides large utility improvement.

Table 6.3: Area resources, clock period, and performance of generated IPv4/IPv6 header processor using different DSE algorithms

DSE algorithm	LUTs	Regs	DSPs	Clock period (ns)	Performance (Million packets per second)	Generated configuration
Gorilla++ rule-based refinement	39,526	57,752	0	7.5	53	12 IP engines each with three threads, pipelined and offloaded IPv4 lookup, IPv6 lookup, and QoS counter
Hill climbing	60,957	84,983	0	8.3	42	10 IP engines each with six threads, pipelined and offloaded IPv4 lookup, IPv6 lookup, and QoS counter
Simulated annealing	39,526	57,752	0	7.5	53	12 IP engines each with three threads, pipelined and offloaded IPv4 lookup, IPv6 lookup, and QoS counter

6.4 MPLS header processor

An MPLS switch is also a networking device that determines the output port to which the incoming packets should be forwarded. There are three main differences between an MPLS switch and a IP router. First, MPLS is a lower-level protocol and as the result it can encapsulate any layer 3 protocol.[4] Second, since MPLS labels are 20 bits (compared to 32-bit IPv4 address and 128-bit IPv6 address) the lookup process is simpler. Third, MPLS is designed with the intrinsic support for stacking the labels, e.g., for tunneling the packets through transient networks.

Figure 6.7 shows the main processing steps of an MPLS header processor. After the *Dispatch* and *Ethernet* steps, the label(s) are processed. The TTL (time-to-live field) for each label is checked in the *MPLS check* step. The

[4]The MPLS header is often placed between the layer 2 and layer 3 protocol headers. Therefore, it is referred to as a layer 2.5 protocol.

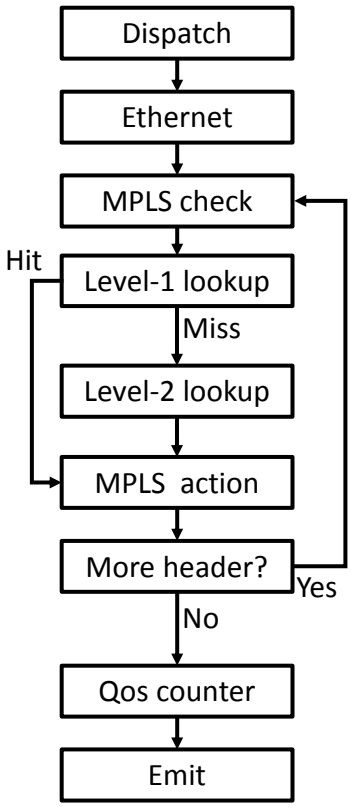


Figure 6.7: Processing steps for the MPLS header processor.

label is looked up in the *Lookup level 1* and *Lookup level 2* processing steps and based on the result of the lookup, the *MPLS action* step decides either (i) to swap a label with a new one or (ii) to keep the label intact. The lookup process also determines if consequent labels should be investigated or not. Based on the result of the last lookup, the output port that the packet should be sent to is determined. At the end, a QoS counter is updated and the updated header is emitted to the output.

For each MPLS packet, up to four lookups are performed. Like our previous work [51], we use a two-level lookup architecture. The first-level lookup is a two-level trie-based algorithm. The 20-bit label is split into two parts; the 16 higher bits are associated to level 1 and the four lower bits are associated to level 2. The lookup data for each trie-level are stored in a separate on-chip scratch-pad memory. If due to the collision in the table entry the first-level lookup is not successful, the second-level lookup memory is used. Like IP header processor, QoS counter requires six accesses to memories, including three read accesses and three write accesses. We used 64-byte packets with random numbers of tags between one and four as the test inputs for MPLS header processor.

6.4.1 Base micro-architecture

The Gorilla++ composition code of the MPLS header processor is shown in Figure 6.8. The corresponding base micro-architecture is shown in Figure 6.9. The **MPLS engine** is connected to eight scratch-pad memories for the four lookup onloaded functions. The **second-level lookup** and **QoS counter** uses off-chip QDR memories. Similar to IPv4/IPv6 header processor, each off-chip QDR memory is modeled as three scratch-pad memories. The on-

```
class Top extends gComponent with include {
  val mplsEngine =  Engine("mplsEngine.c")
  val mems = for (i <- 0 until 8) {
   yield spMem(height=1024, width=32, "MemInitialConent" + i)
  } ++ for (i <- 9 until 11) {
   yield spMemRW(height=1024, width=64, "MemInitialConent" + i)
  }
  val qdr1Mem = mems.slice(0, 1)
  val qdr2Mem = mems.slice(2, 3)
  val qdr3Mem = mems.slice(4, 5)
  val qdr4Mem = mems.slice(6, 7)
  val qdr5Mem = mems(8)
  val memOffloads = ArrayBuffer((qdr1Mem(0), "lookup1Mem1"),
   (qdr1Mem(1), "lookup1Mem2"), (qdr2Mem(0), "lookup2Mem1"),
   (qdr2Mem(1), "lookup2Mem2"), (qdr3Mem(0), "lookup2Mem1"),
   (qdr3Mem(2), "lookup3Mem2"), (qdr4Mem(0), "looku4Mem1"),
   (qdr4Mem(1), "lookup4Mem2"), (qdr5Mem, "secondLevelLookupMem"),
   (qdr4Mem(0), "qosMem1"), (qdr4Mem(1), "qosMem2"),
   (qdr4Mem(2), "qosMem2"))
  val result = Offload(mplsEngine, memOffloads)
}
```

Figure 6.8: Composition code for an MPLS header processor.

chip scratch-pad memory for the second-level lookup has 1,024 entries. Each entry is associated with 1,024 consequent entries in the actual QDR memory. We use a synthetic lookup table and a test input traffic that does not cause any misses in the simplified second-level lookup.

6.4.2 Refined micro-architecture

Gorilla++ refines the base micro-architecture of the MPLS header processor in 20 iterations. The refined micro-architecture consists of 11 replicated **MPLS engine**s. Each engine has five execution threads. The engines are

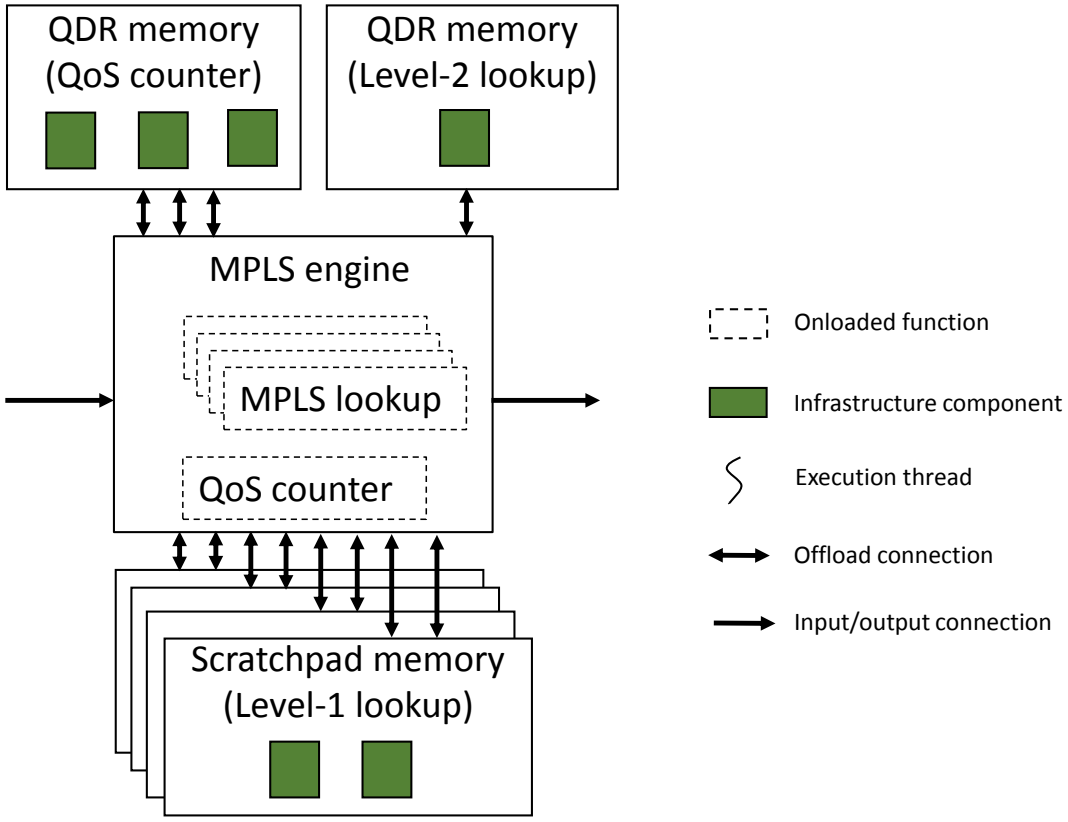


Figure 6.9: MPLS header processor base micro-architecture.

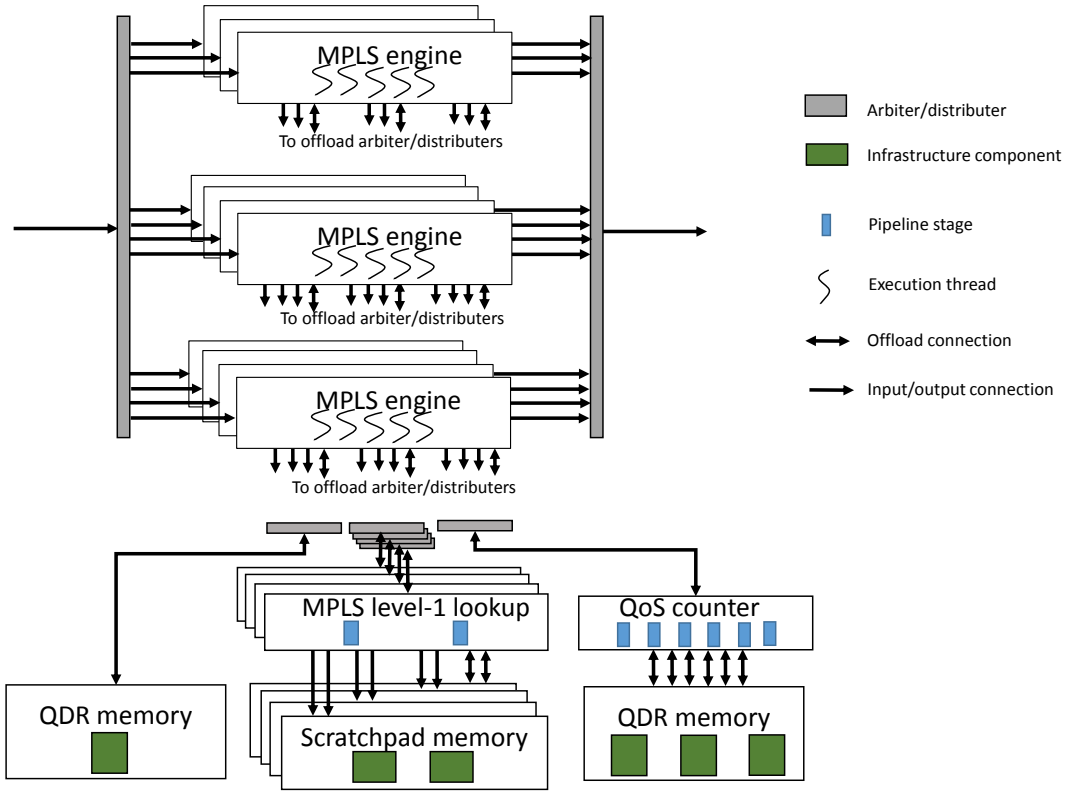


Figure 6.10: MPLS header processor refined micro-architecture.

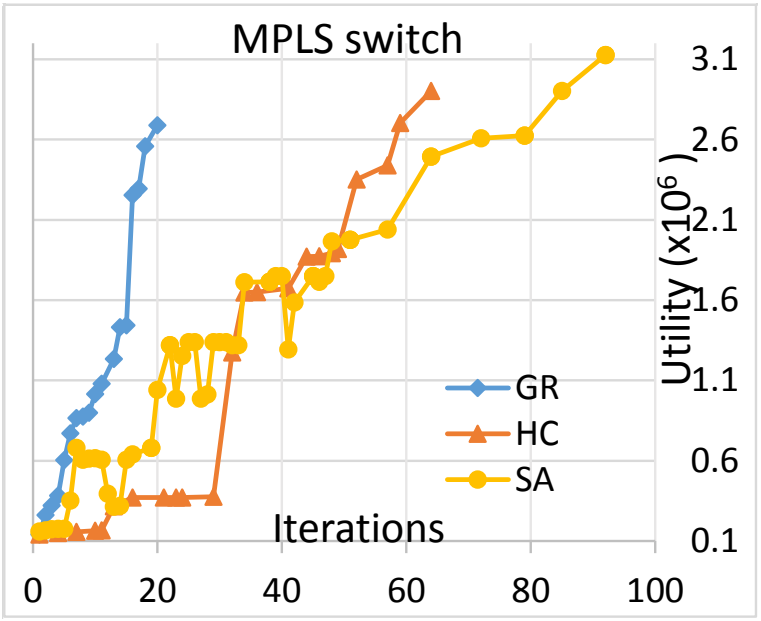


Figure 6.11: DSE of the MPLS header processor (i) HC:hill climbing, (ii) SA:simulated annealing, and (iii) GR:Gorilla++ refinement algorithm.

connected to pipelined and offloaded **level-1 lookup**s. The engines are also connected to a pipelined and offloaded **QoS counter** as well as the **level-2 lookup** memory.

The parameters of the manually parameterized version of the composition code are shown in Table 6.4. The result of exploring the design space using generic optimization algorithms as well as the Gorilla++ rule-based DSE is shown in Figure 6.11. Also, Table 6.5 shows the FPGA resources, clock frequency, and performance of the generated accelerators.

Unlike the IPv4/IPv6 header processor, hill climbing generates lower area than Gorilla++. This is due to the fact that MPLS lookups are simpler and therefore require a small area. Therefore, offloading them generates more

Table 6.4: Parameters in the manually parameterized MPLS header processor

Parameter	Type	Range	Initial value
MPLS lookup1 offload type	offload type	onloaded, offloaded, pipeOffloaded	onloaded
MPLS lookup2 offload type	offload type	onloaded, offloaded, pipeOffloaded	onloaded
MPLS lookup3 offload type	offload type	onloaded, offloaded, pipeOffloaded	onloaded
MPLS lookup4 offload type	offload type	onloaded, offloaded, pipeOffloaded	onloaded
QoS counter offload type	offload type	onloaded, offloaded, pipeOffloaded	onloaded
Number of MPLS engine threads	numeric	1-8	1
Number of MPLS engines	numeric	1-16	1

overhead associated with a separate, stand-alone engine infrastructure than the area saving associated to sharing the core lookup logic.[5] While simulated annealing keeps all the lookups as onloaded functions, the hill climbing offloads two of the lookups.

[5]Currently, the refinement algorithm assumes that pipe-offloading is always result in a lower area that a non-offloaded version. Therefore, its criteria is solely checks if this refinement provides the desired performance. A more accurate criteria that contains an area model for evaluating the effectiveness of the pipe-offloading can resolve this problem.

Table 6.5: Area resources, clock period, and performance of generated MPLS header processor using different DSE algorithms

DSE algorithm	LUTs	Regs	DSPs	Clock period (ns)	Performance (Million packets per second)	Generated configuration
Gorilla++ rule-based refinement	56,884	61,564	0	7.9	59	11 MPLS engines each with six threads, pipelined and offloaded lookup1, lookup2, lookup3, lookup4, and QoS counter
Hill climbing	38,432	52,341	0	7.6	61	14 MPLS engines each with four threads, onloaded lookup1 and lookup2, pipelined and offloaded lookup3, lookup4, and QoS counter
Simulated annealing	42,873	57,198	0	7.7	65	18 MPLS engines each with three threads, onloaded lookup1 and lookup2, lookup3, and lookup4, pipelined and offloaded QoS counter

6.5 Parallel K-means

K-means is an iterative data-mining algorithm for clustering n points in d dimensional Euclidean space into k clusters. Each cluster at any given iteration of the algorithm has a corresponding centroid. Initial centroids can be determined randomly.

In each iteration, (i) for a given point, the closest centroid is found and the point is associated to the corresponding cluster and (ii) new centroids are calculated by averaging the points in each cluster. The new centroids are used as the inputs to the next iteration of the algorithm. The algorithm finishes when the association of points to clusters is not changed across iterations or the maximum number of iterations is reached.

K-means is often executed for a large number of points (n is a big

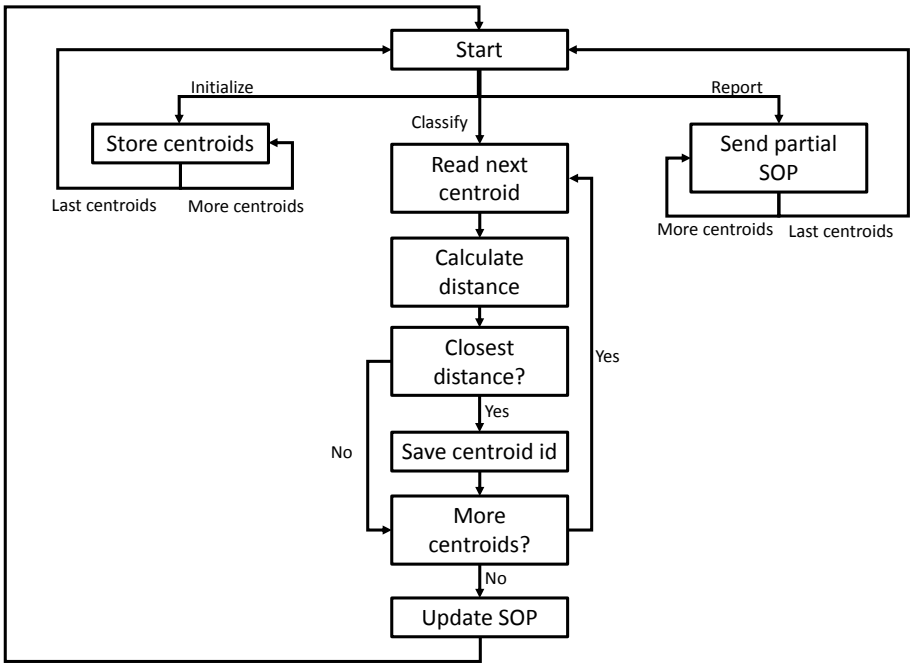


Figure 6.12: Processing steps for the K-means accelerator.

number). K-means can be parallelized to improve the throughput of clustering the points and consequently to reduce the execution time of each iteration. One parallelize solution to run K-means on multiple processors is to copy the centroids to each processor and also to have an array of partial sum-of-points (SOP) values in each processor [42] (one array entry for each centroid). In this solution, we can (i) assign a point to an arbitrary processor, (ii) find the closest centroid to the point, and (ii) add the point's coordinates values to the SOP entry. After all points are clustered, a reduction phase generates the new centroids from the partial SOP arrays from different processors.

Figure 6.12 shows the processing steps of a **kEngine**, an engine that is acting similarly to one of the processors in the above parallel K-means algorithm. The engine has three computation phases: (i) *initialize*, (ii) *calculate*, (iii) *report*. The first and third phases are for collecting the old centroids from input and sending the SOPs to the output respectively. In the second phase, when a new point is received, the engine iterates over all centroids and *calculates the distance* between the point and the centroid. If the centroid is the closest centroid to the point, the distance to the centroid and the id of the centroid are saved. After visiting all centroids, the engine *updates the corresponding SOP entry*. Also, the corresponding number of points that are associated with the centroid is updated. The engine is ready to receive and cluster a new point at this stage. Our experiments are based on clustering 3-dimensional double-precision points into 20 clusters.

6.5.1 Base micro-architecture

Figure 6.13 shows the composition code for the K-means accelerators. **kePlus** includes the **kEngine**, centroid memory, SOP memory, and cluster

counts memory. A map-reduced, replicated composition is used to parallelize the accelerator by having multiple instances of the **kePlus**. We specified the user-defined mapper and user-defined reducer function for the map-reduced replication. The mapper broadcasts the incoming centroids to the **kEngines** in the first phase of the algorithm. However, it sends an incoming point to only to one of the **kEngines** in the second phase. In the second phase, the clustering of points is done. The **kEngines** use a lock component, **kLock**, to ensure mutual exclusion when updating the shared scratch-pad memories (SOP memory and cluster counts memory) by multiple **kEngine** threads or multiple **kEngines**. After points are clustered during the second phase, during the third phase of the computation, the **kEngines** send their SOP and cluster counts values to the reducer component. The reducer aggregates all SOP values and cluster counts in its own local memories and generates the new set of centroids.

6.5.2 Refined micro-architecture

Gorilla++ refines the micro-architecture of the K-means accelerator in nine iterations. The refined micro-architecture contains four **kePluses**, each with one **keEngine**. Each **kEngines** has three threads. The micro-architecture contains two pipelined distance calculators. Each distance calculator is shared by two kePluses.

The parameters of the manually parameterized version of the composition code are shown in Table 6.6. The result of exploring the design space using generic optimization algorithms as well as Gorilla++ rule-based DSE is shown in Figure 6.16. Also, Table 6.7 shows the FPGA resources, clock frequency, and performance of the generated accelerators. Hill climbing uses Four

```
class Top extends Component with include {
  val add  = fpAdder()
  val mul  = fpMultiplier()
  val div  = fpDivider()
  val sqrt = fpSqrt()
  val centroidsMem = spMem(height = NUM_OF_CENTROIDS,
            width=192);
  val SOPMem = spMem(height = NUM_OF_CENTROIDS,
            width = 192);
  val clusterCountsMem = spMem(height = NUM_OF_CENTROIDS,
            width = 32);
  val kEngine =  Engine("kEngine.c")
  val kMapper =  Engine("kMapper.c")
  val kReducer =  Engine("KReduce.c")
  val kePlus = Offload(kEngine,
   ArrayBuffer((centroidsMem, "centroidsMem"),
   (SOPMem, "SOPMem"),
   (clusterCountsMem, "clusterCountesMem"),
   (mul, "mul"), (add, "add"), (sqrt, "sqrt"))
  val krPlus = Offload(kReducer,
   ArrayBuffer((SOPMem, "SOPMem"),
   (clusterCountsMem, "clusterCountsMem"),
   (div, "div", (add, "add")))
  val result = Replicate(kePlus, NUM_OF_KEPLUSES,
                distributor = kMapper,
                aggregator = krPlus)
}
```

Figure 6.13: Gorilla++ composition code for a K-means accelerator.

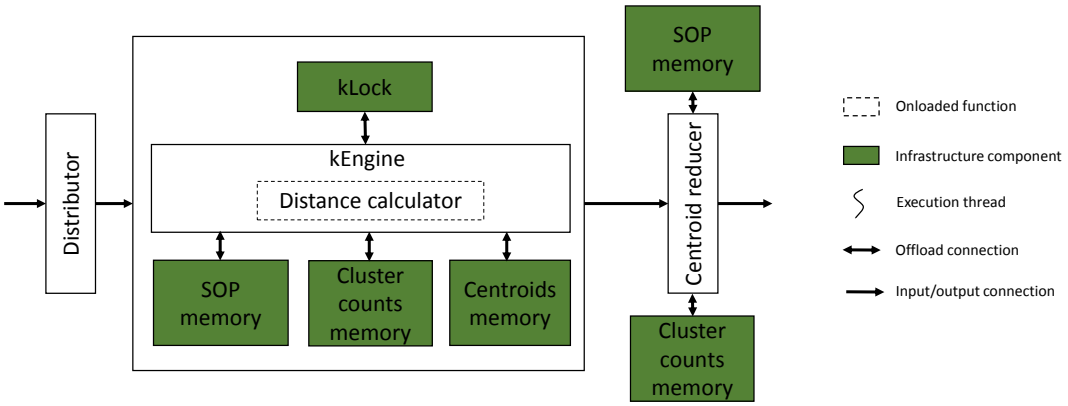


Figure 6.14: K-means accelerator base micro-architecture.

Table 6.6: Parameters in the manually parameterized K-means accelerator

Parameter	Type	Range	Initial value
Distance calculator offload type	offload type	onloaded, offloaded, pipeOffloaded	onloaded
Distance calculator sharing factor	sharing factor	1-4	1
Number of kEngine threads	numeric	1-8	1
Number of kEngines	numeric	1-8	1
Number of kePlus components	numeric	1-8	1

kePluses, each with three **kEngines**. Each **KEngine** has two threads. Simulated annealing uses six **kePluses** each with one **kEngine**. Each **kEngine** has two threads.

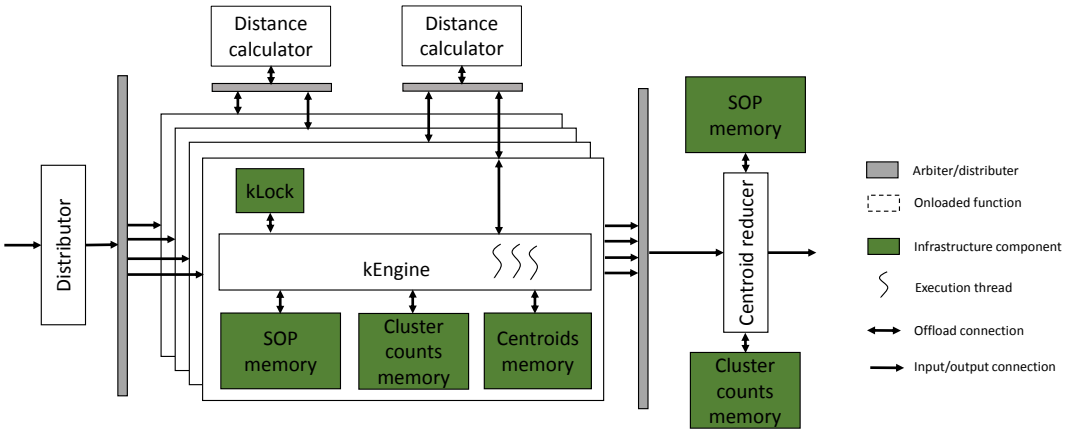


Figure 6.15: K-means accelerator refined micro-architecture.

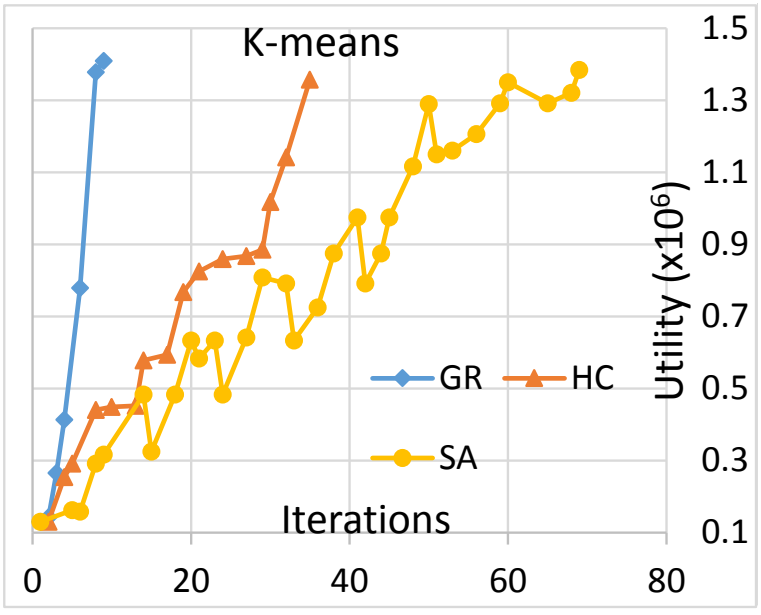


Figure 6.16: DSE of the K-means accelerator using (i) HC:hill climbing, (ii) SA:simulated annealing, and (iii) GR:Gorilla++ refinement algorithm.

Table 6.7: Area resources, clock period, and performance of generated K-means accelerator using different DSE algorithms.

DSE algorithm	LUTs	Regs	DSPs	Clock period (ns)	Performance (Million points per second)	Generated configuration
Gorilla++ rule-based refinement	37,852	45,252	99	7.5	0.66	Four kePluses, one kEngine with three threads, pipelined and offloaded distance calculator, distance calculator sharing factor: two
Hill climbing	77,738	85,325	99	7.7	0.66	Four kePluses, three kEngines with two threads, pipelined and offloaded distance calculator, distance calculator sharing factor: two
Simulated annealing	35,654	44,765	99	7.5	0.66	Six kePluses, one kEngine with two threads, pipelined and offloaded distance calculator, distance calculator sharing factor: two

6.6 Parallel PageRank

PageRank is an iterative algorithm that is used to determine the popularity of a web page on the Internet. This is done by calculating the probability of referring to the page on a given random access. The pages that are referenced more from other pages are considered higher in rank. In each iteration, the algorithm goes through each page and distributes the current rank of the page evenly across all referenced pages in form of new updates. At the end of the iteration, all updates from referencing pages are summed up to form the new rank value for the referenced page. Figure 6.20 shows the composition code for the PageRank accelerator.

Figure 6.19 shows the layout of the PageRank data structure we used in our implementation. The data structure consists of four arrays. The first array contains the page information, including the number of links, as well as the

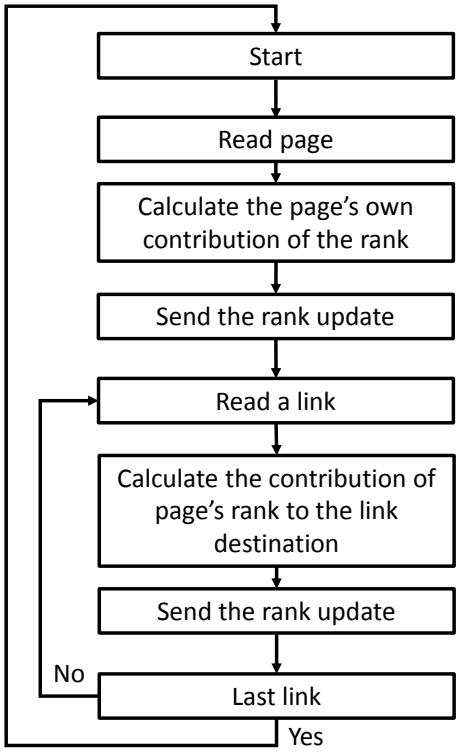


Figure 6.17: Processing steps for the PageRank update generator.

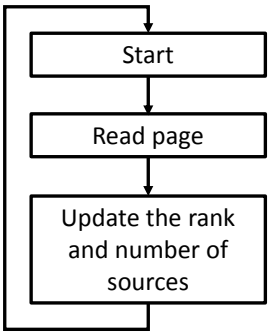


Figure 6.18: Processing steps for the PageRank update writer.

pointer to the first and last outgoing links from the page to other pages. The second array contains the link pointers. The third and fourth arrays contain two versions of the rank information for each page. At any iteration of the algorithm, one of the two arrays is used as the old ranks and the other one is used as the new ranks. At the end of the iteration, the role of the two ranks arrays are switched.

One way to parallelize the PageRank [42] is to divide the pages between different processors. Each processor is responsible (i) to propagate the updates from its own pages to the other pages and (ii) to receive and apply the updates from other pages to its own pages. This can be done by splitting the job of each processor to two independent tasks (i) the update generator and (ii) the update writer. The update generator works with the old instance of the rank values while the update writer works with the new instance of the rank values. The instances are swapped after finishing each iteration. In our implementation, we used pages with random fan-out between 1 and 15 and double-precision rank values to test the PageRank accelerator.

6.6.1 Base micro-architecture

Inspired by the above parallelization method, the micro-architecture of the PageRank accelerator consists of two chained components, the **update generator** and the **update writer**. The **update generator** walks through the pages in the memory, and for each page, (i) finds its outgoing links and (ii) generates the contribution of the rank update values from the page. Both of these components are connected to global DRAM through two private caches in order to access PageRank data structure. The caches are flushed after each iteration of the PageRank algorithm – ensuring that the old and new rank

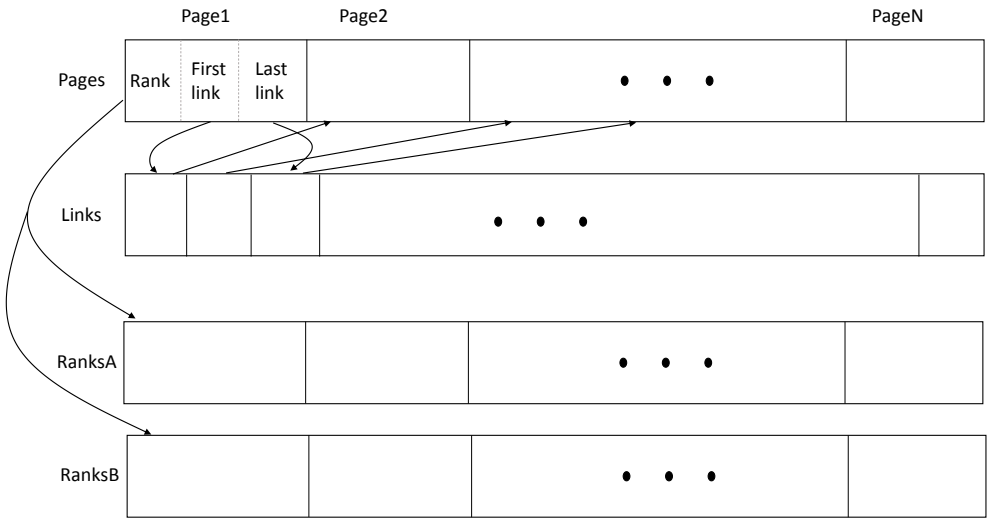


Figure 6.19: PageRank data structure.

values are read from the DRAM at the next iteration. The update writer uses a lock engine, **uwLock**, to make sure that the rank values are updated atomically.

6.6.2 Refined micro-architecture

Gorilla++ refines the micro-architecture of the PageRank accelerator in 21 iterations. The refined micro-architecture contains two **update generators**, each with seven threads, and two **update writers**, each with six threads. The **update calculator** is offloaded and pipelined. The caches use 5 threads to improve memory-level parallelism.

The parameters of the manually parameterized version of the composition code are shown in Table 6.8. The result of exploring the design space using generic optimization algorithms as well as the Gorilla++ rule-based DSE is shown in Figure 6.23. Also, Table 6.9 shows the FPGA resources, clock

```
class Top extends Component with include {
  val ug =  Engine("updateGenerator.c")
  val uw = Engine("updateWriter.c")
  val cache =
   Cache(height=1024, lineSize=128, tagSize=20, threads=1)
  val uwLock = lock(numOfLocks = 1000)
  val add = FPDPAdder()
  val div = FPDPDivider()
  val ugPlus = Offload(Offload(ug, div, "div"),
   (cache, "mem"))
  val uwPlus = Offload(uw, ArrayBuffer((uwLock, "lock",
  (adder, "adder"), (cache, "mem"))
  val result = Offload(Chain(ugPlus, uwPlus),
                       (DRAM, "dram"))
}
```

Figure 6.20: Gorilla++ composition code for a PageRank accelerator.

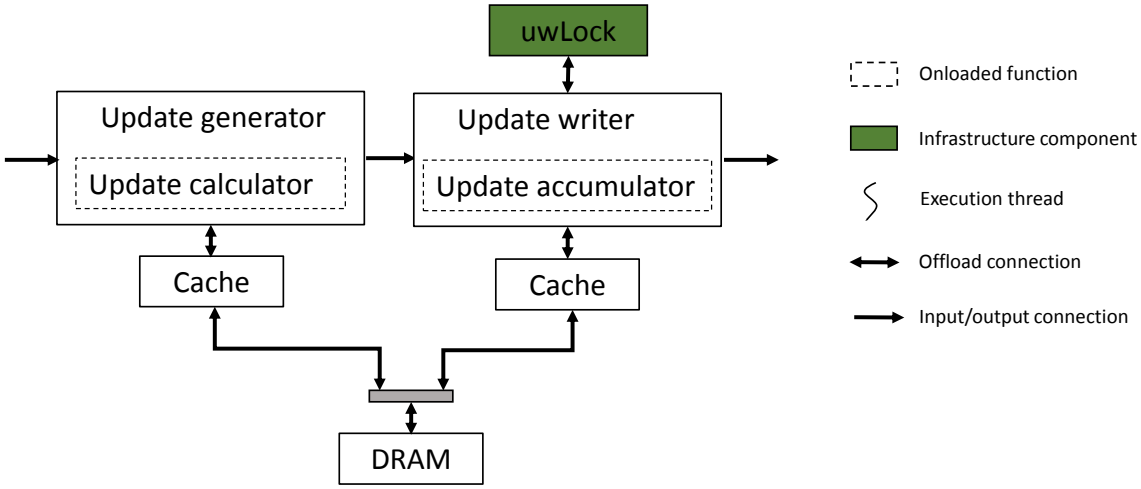


Figure 6.21: The base micro-architecture of the PageRank accelerator.

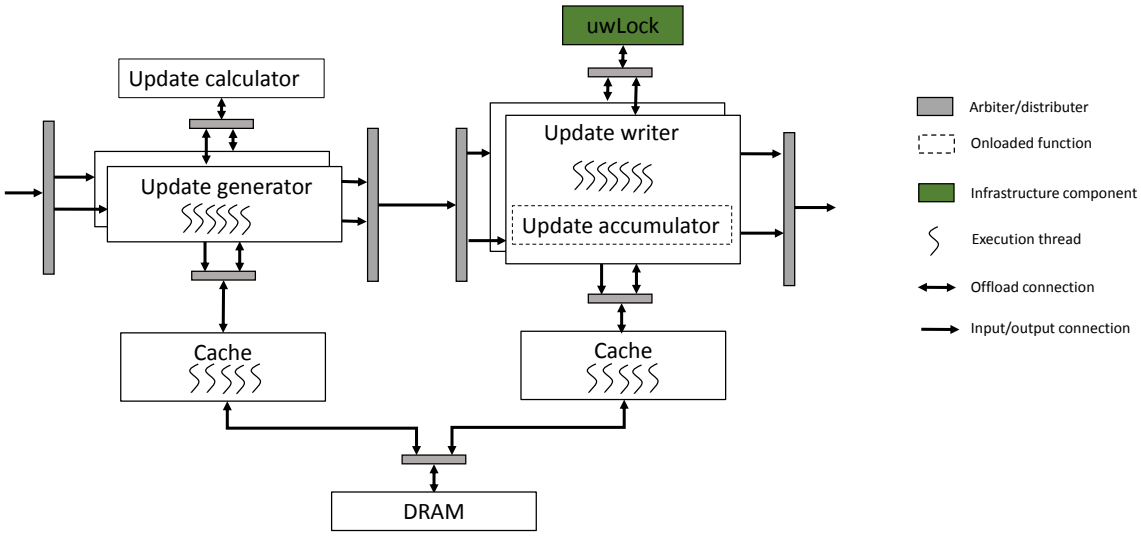


Figure 6.22: The refined micro-architecture of the PageRank acceleration.

frequency, and performance of the generated accelerators. Hill climbing uses more replication than multi-threading comparing to Gorilla++. Simulated annealing uses multi-threading more than hill climbing.

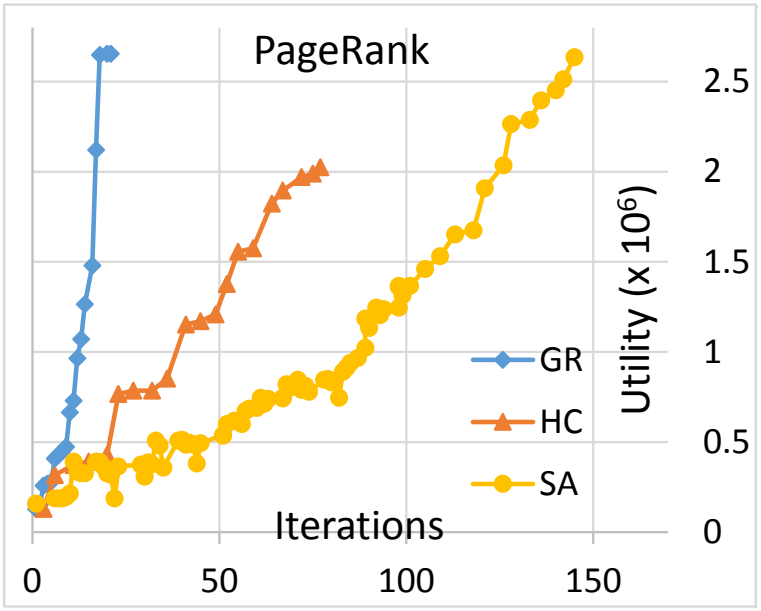


Figure 6.23: DSE of the PageRank accelerator using (i) HC:hill climbing, (ii) SA:simulated annealing, and (iii) GR:Gorilla++ refinement algorithm.

Table 6.8: Parameters in the manually parameterized PageRank accelerator

Parameter	Type	Range	Initial value
Update calculator offload type	offload type	onloaded, offloaded, pipeOffloaded	onloaded
Update accumulator offload type	offload type	onloaded, offloaded, pipeOffloaded	onloaded
Floating point adder sharing factor	sharing factor	1-4	1
Number of update generator threads	numeric	1-8	1
Number of update generator engines	numeric	1-8	1
Number of update writer threads	numeric	1-8	1
Number of update writer engines	numeric	1-8	1
Number of cache threads	numeric	1-8	1

Table 6.9: Area resources, clock period, and performance of generated PageRank accelerator using different DSE algorithms

DSE algorithm	LUTs	Regs	DSPs	Clock period (ns)	Performance (Million pages per second)	Generated configuration
Gorilla++ rule-based refinement	18,813	17,872	22	10.99	0.031	Two update generators each with six threads, two update writers each with seven threads, pipelined and offloaded update calculator, onloaded update accumulator, adder sharing factor: two, number of cache threads: five
Hill climbing	19,169	18,078	22	11.1	0.028	Three update generators each with four threads, three update writers each with four threads, onloaded update calculator, onloaded update accumulator, adder sharing factor: two, number of cache threads: five
Simulated annealing	20,245	18,960	22	10.99	0.031	Three update generators each with four threads, two update writers each with seven threads, pipelined and offloaded update calculator, onloaded update accumulator, adder sharing factor: two, number of cache threads: five

6.7 Memcached

Memcached [60] is an open source key-value system for in-memory objects which is often used as an application-level cache for conventional data-repository systems [62]. Distributed Memcached clients use a consistent hashing scheme to find the home server of an object of interest. The server caches frequently requested objects in a hash table and uses its own hash function to find the object. Memcached has a number of commands for manipulating the objects (e.g., *get*, *set*, *delete*, etc.) with a nearly 30-to-1 ratio of *get* to other types of requests in real workloads [3].

We proposed a technique [50] to slice the Memcached application into a fast-path to handle frequently used *g*et commands and a slow-path to handle the other commands. The fast-path was implemented on an FPGA and the slow-path was implemented on general-purpose cores. The Memcached fast-path is used as one of the Gorilla++ case studies. The white boxes in Figure 6.24 show the processing steps of the accelerator. The accelerator receives the request packets, parses them, and determines if they belong to the fast-path. If so, the accelerator *finds the corresponding item* in the global hash data structure in the memory and *sends the item data* back to the client. Lastly the fast-path performs necessary *clean-up* to finish the processing of the request.

6.7.1 Base micro-architecture

Figure 6.26 shows the composition code for the Memcached accelerator and Figure 6.27 shows the corresponding base micro-architecture. The processing of incoming requests is done by **mcfpEngine**. The **mcfpEngine** accesses the hash table data structure through a cache and uses a lock, **mclock**, to

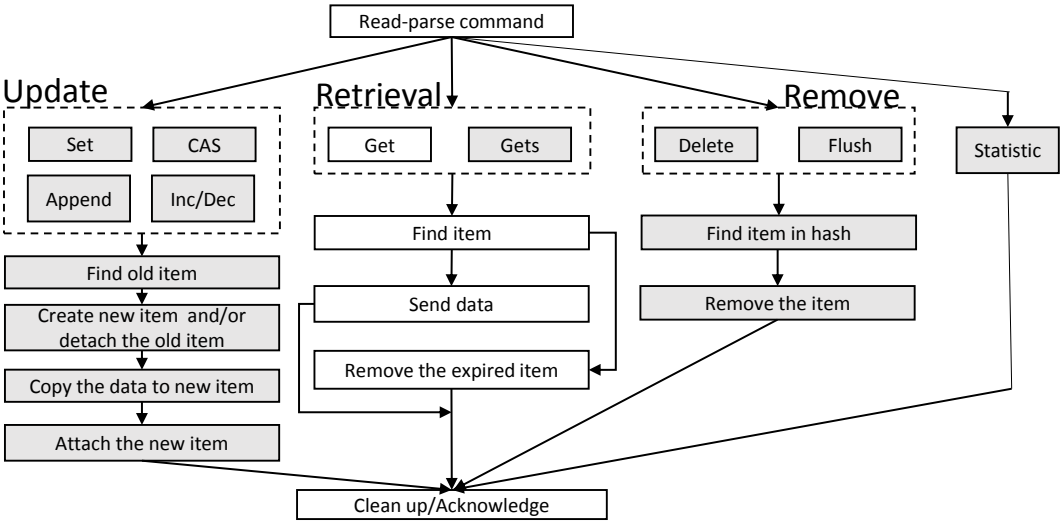


Figure 6.24: High-level flow-chart of Memcached server application. The white boxes form the Memcached fast-path, which is implemented as an accelerator.

provide mutual exclusion when accessing an item.

Figure 6.25 shows the layout of a Memcached data structure. The Memcached items are accessible through a hash structure. A hash function is used to find the item's bucket, which contains a link-list of the items with the same hash value. Each item contains several meta data, as well as the actual data associated with the item.

Each item is also accessible through a slab allocation data structure, which is a user-level memory allocation and garbage collection mechanism. Different item sizes belong to different slab allocation classes. Upon accessing an item, the item is moved to the head of a least-recently-used (LRU) list which is used by the slab allocator for evicting the items. To test the Memcached, we populated a software-based Memcached with one million 64-byte items and then used the contents of the memory as the function model of the DRAM in

Table 6.10: Parameters in the manually parameterized Memcached fast-path accelerator

Parameter	Type	Range	Initial value
Hash function offload type	offload type	onloaded, offloaded, pipeOffloaded	onloaded
Number of MCFPEngine threads	numeric	1-8	1
Number of MCFPEngines	numeric	1-8	1
number of cache threads	numeric	1-8	1

our simulation.

6.7.2 Refined micro-architecture

Gorilla++ refines the micro-architecture of the fast-path Memcached accelerator in 11 iterations. The refined micro-architecture contains five **mcfpEngine**s, each with three threads. The hash function is pipelined and offloaded. Two hash functions are used in the generated micro-architecture, each shared by three **mcfpEngine**s. The cache uses five threads to improve memory-level parallelism.

The parameters of manually parameterized version of the composition code is shown in Table 6.10. The result of exploring the design space using generic optimization algorithms as well as Gorilla++ rule-based DSE are shown in Figure 6.29. Also, Table 6.11 shows the FPGA resources, clock frequency, and performance of the generated accelerators. Hill climbing uses more threads in **mcfpEngine**s than Gorilla++. Simulated annealing uses more engines and fewer number of threads – resulting in higher overall area.

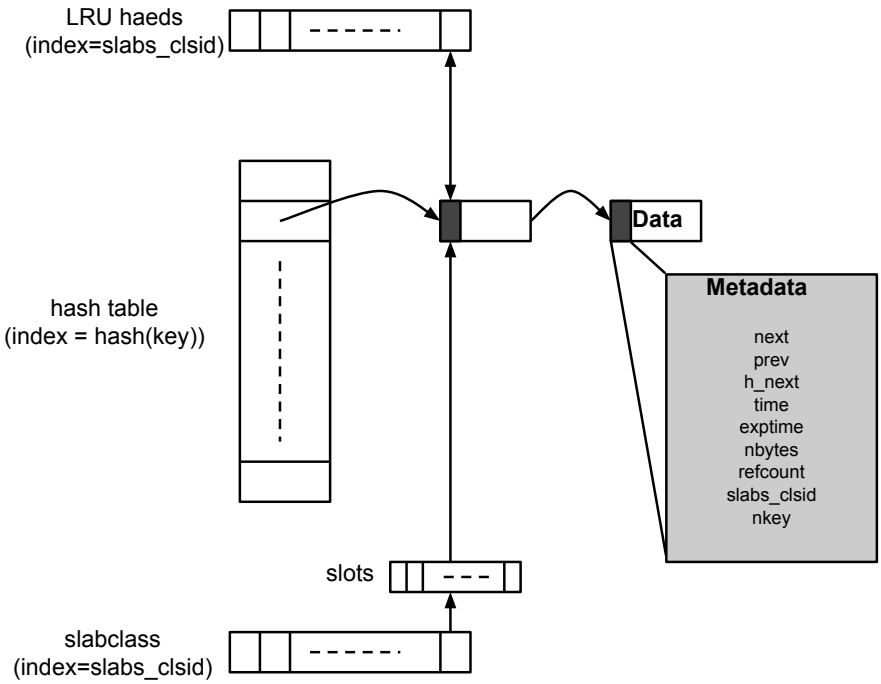


Figure 6.25: Memcached data structure. **next** and **prev** are pointers for LRU-list. **h_next** is the pointer to the next item in the hash-table link-list. **time** is the last time an item is accessed. **exptime** is the time that an item should be expired. **nbytes** is the size of the data associated with the item. **refcount** is the number of concurrent threads that are currently working with the item. **slabs_clsid** is the slab class id associated with the item. **nkey** is the size of the item's key.

```
class Top extends Component with include {
  val mcfpe = Engine("mcfpEngine.c")
  val cache =
   Cache(height=1024, lineSize=128, tagSize=20, threads=1)
  val mem = Offload(cache, DRAM)
  val mcLock = lock(numOfLocks=1000)
  val result = Offload(mcpe,ArrayBuffer(mem, mcLock))
}
```

Figure 6.26: Gorilla++ composition code for a Memcached fast-path accelerator.

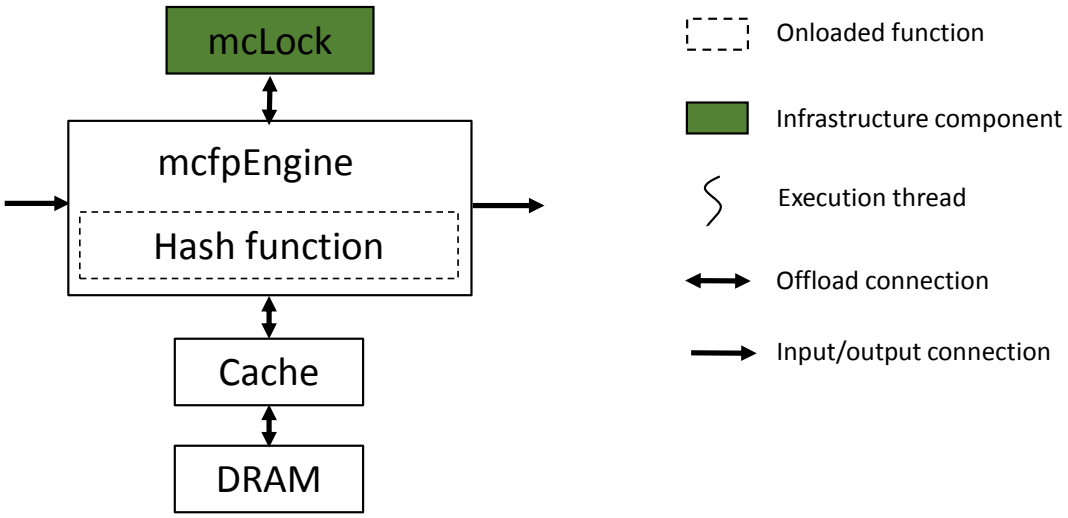


Figure 6.27: The base micro-architecture of the Memcached fast-path accelerator.

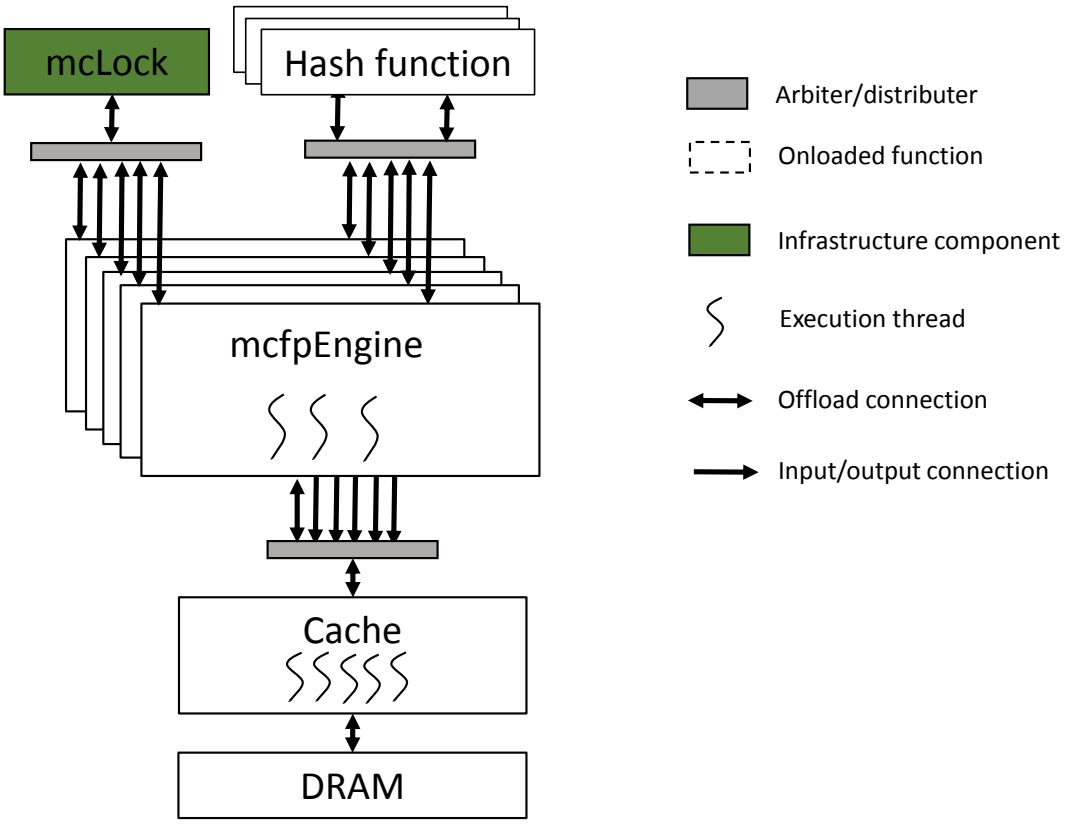


Figure 6.28: The refined micro-architecture of the Memcached fast-path accelerator.

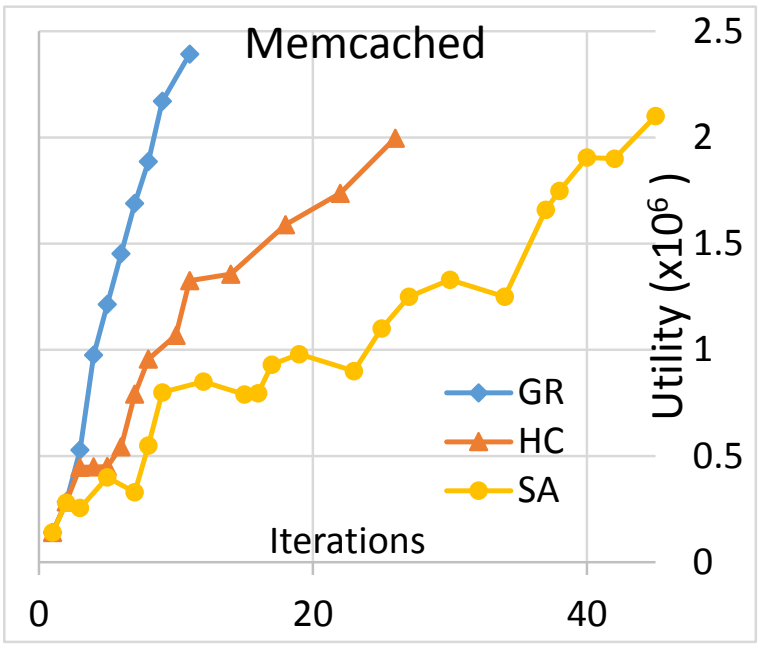


Figure 6.29: DSE of the Memcached fast-path accelerator using (i) HC:hill climbing, (ii) SA:simulated annealing, and (iii) GR:Gorilla++ refinement algorithm.

Table 6.11: Area resources, clock period, and performance of Memcached fast-path accelerator using different DSE algorithms

DSE algorithm	LUTs	Regs	DSPs	Clock period (ns)	Performance (Million Gets per second)	Generated configuration
Gorilla++ rule-based refinement	37,698	28,748	0	9.5	3.5	Five mcfpEngines each with three threads, offloaded hash function with replication factor of three, number of cache threads: five
Hill climbing	54,587	40,921	0	10	3.2	Five mcfpEngines each with five threads, offloaded hash function with replication factor of three, number of cache threads: six
Simulated annealing	38,765	29,187	0	9	3.11	Seven mcfpEngines each with two threads, offloaded hash function with replication factor of two, number of cache threads: five

6.8 Gorilla++ language vs. Chisel

For the benchmark applications presented in this section, Gorilla++ descriptions have on average 3.9x less LoC comparing to the generated Chisel (Table 6.1). Through inspecting the generated Chisel code, we found the quality of the generated code close to the quality of a hand-written code. Note that the Gorilla++ programming model consists of C-kernels and regulated composition constructs – making it attractive for programmers with less hardware experience.

6.9 Compiler optimization results

Replication and pipelining are two refinements that are not specific to Gorilla++. Although for a smaller class of applications, Cong et al., among others, studied these refinements [20] for synchronous dataflow HLS. Since

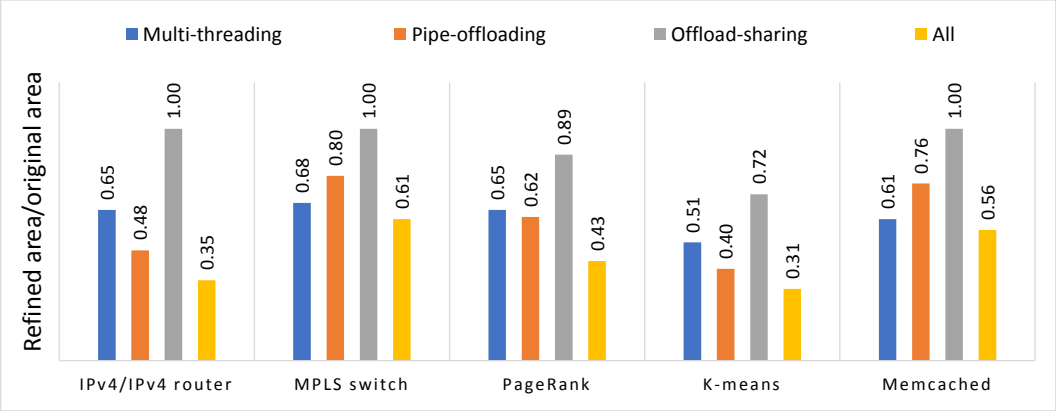


Figure 6.30: The impact of major Gorilla++-specific refinements on the area of the generated accelerators.

Gorilla++ uses offload interfaces as a first order construct in the execution model and language, the refinements associated with the offloading are specific to Gorilla++.[6] These refinements include multi-threading, offloading, pipe-offloading, thread removal, and offload-sharing. We should also note that both replication and pipelining are more sophisticated than their SDF-based equivalents, since they support components with offload interfaces.

Figure 6.30 shows the effect of these refinements on the area of generated accelerators. These effects are measured by first determining the area[7] of a Gorilla++-refined accelerator when all the refinements except replication is inactive. Then, each refinement is activated and the corresponding area is measured. Last, the accelerator area when all Gorilla++ refinements are active is measured. Since when multi-threading or pipe-offloading is inactive,

[6] As discussed in Chapter 4, other researchers are also supported multi-threading in HLS but there are differences between Gorilla++ and their work.

[7] The area is the sum of LUT utilization and DSP utilization.

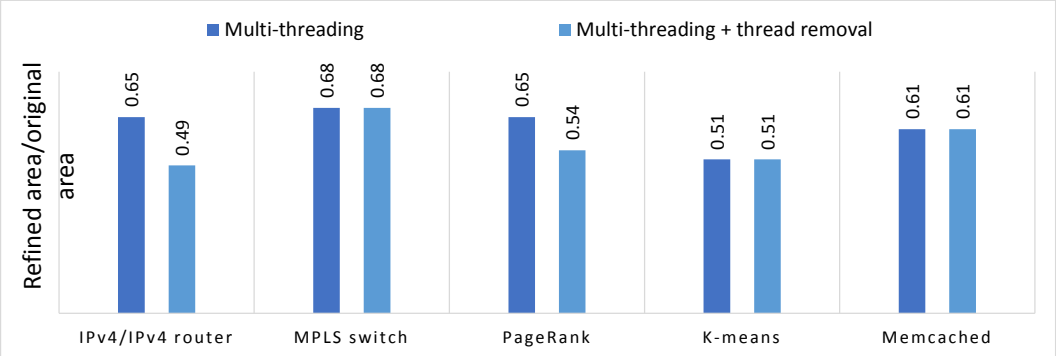


Figure 6.31: The impact of the thread removal refinement on the area of the generated accelerators.

replication can still improve the throughput, in all these scenarios, the same throughput, albeit with different area, is achieved. The aggregated area saving when all these three refinements are active is less than the sum of their individual savings. This is because the resources that these refinements are sharing are not completely disjoint. For example, pipe-offloading is a hybrid refinement that combines (i) multi-threading, (ii) offloading, and (iii) pipelining refinements. Therefore, when measuring the area saving of pipe-offloading and multi-threading in isolation, part of the area saving is the result of sharing the same resources in the design. Similarly, although offload-sharing share the offloaded resources, however, multi-threading can share the same resources inside an engine and achieve part of the same benefit. There are however resource sharing that is done only by each one of these refinements and not covered by the two others.

Figure 6.31 demonstrates the effect of thread removal when it is activated along multi-threading refinement. Note that when multi-threading is not active, thread removal cannot be done. Only two of the benchmarks bene-

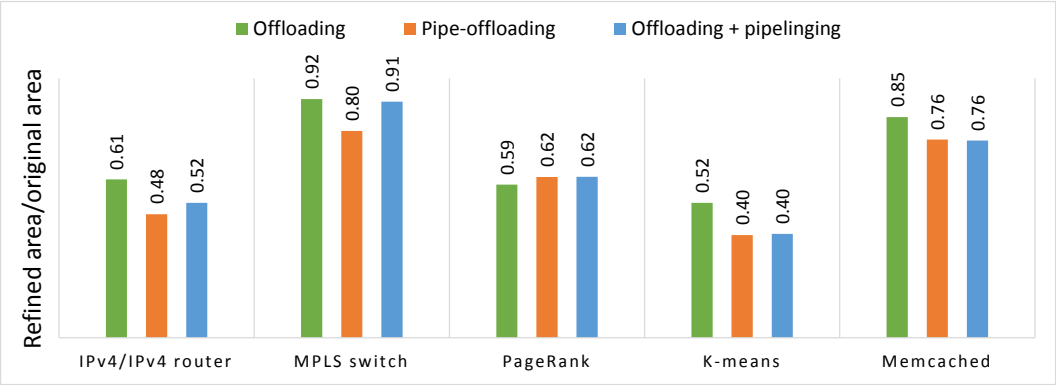


Figure 6.32: The impact of the pipe-offloading, pipelining, and offloading on the area of the generated accelerators.

fit from the thread removal refinement. In both of these cases, the refinement process first uses multi-threading while the offload rate is still high. After applying multi-threading refinements, the offload rate drops and the refinement process turns into the use of the replication refinement. At this stage, the components require more compute resources rather than more contexts to tolerate the offload latencies. However, when replication is used, part of the original performance improvement benefit, associated to the multi-threading refinement, is exploited using the replicating refinement and consequently the thread utilization drops. In this case, thread removal removes under-utilized threads and saves area. Unlike thread removal, the component removal refinement was not activated in any of the Gorilla++ benchmark applications explored in this thesis. However, in order to be able to reverse the effect of replication in general, when the base micro-architecture has a component with a high replication factor, we keep it in the refinement set of Gorilla++.

Figure 6.32 shows the area saving for the cases that either offloading or pipe-offloading is active as well as the case that both offloading and pipelining

are active. Pipe-offloading is more effective than offloading (except one case), since a pipelined and offloaded component has higher throughput. This is important because later in the refinement process, replication might improve the throughput of other components and put more demand on the offloaded components. When both offloading and pipelining are active, their combined benefit is different than the benefit of pipe-offloading (as a single combined refinement). There are cases in IPv4/IPv6 router and MPLS switch that pipe-offloading as a combined refinement is activated but neither the pipelining nor the offloading is activated. This is because in these benchmarks offloading does not provide enough performance improvement to justify the area overhead. However, pipe-offloading provides a higher performance benefit. The onloading refinement is not activated in any of the benchmark applications. However, in order to be able to reverse the effect of the offloading in general, when two under-utilized engines have offloaded relationship, we keep it in the refinement set of Gorilla++.

6.10 DSE results

Figure 6.33 shows the summary of the DSE results in terms of number of DSE iterations as well as the reverse QoR ($1/utility$). On average, Gorilla++ has 3.4x fewer iterations than hill climbing, while generating a hardware with 9% higher utility. Also, on average, Gorilla++ has 6x fewer iterations than simulated annealing, while generating a hardware with 1% lower utility.

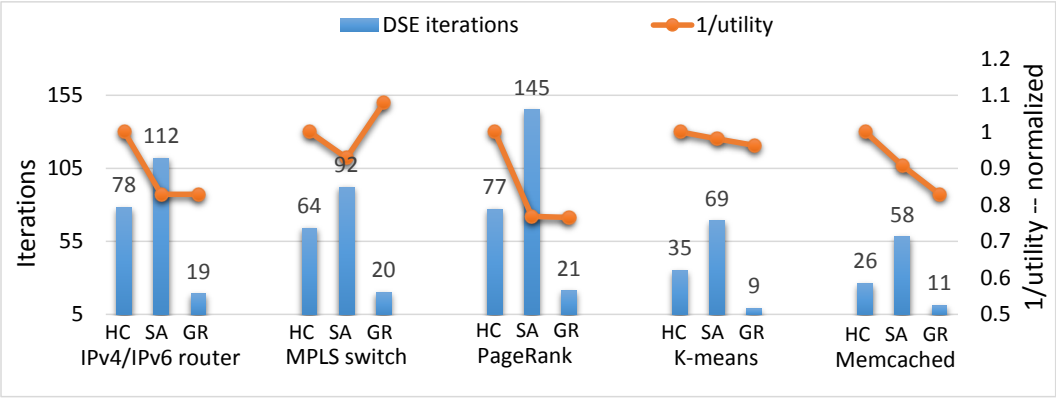


Figure 6.33: Summary of comparisons between (i) HC:hill climbing, (ii) SA:simulated annealing, and (iii) GR:Gorilla++ refinement algorithm.

6.11 Comparison to low-level design methodologies

In this section, we compare three hardware designs generated using Gorilla++ to the manually-crafted designs using low-level hardware description languages. The details of this comparison are presented in table 6.12.

The first design is a non-blocking cache that is part of the open-source LEAP project [89]. The cache is implemented using Bluespec SystemVerilog [8]. The corresponding Verilog model is generated using Bluespec compiler. We implemented a non-blocking cache with the same configuration (direct-mapped, 1024 cache lines, each has a single 32-bits word) in Gorilla++. The Gorilla++ DSE uses four threads to achieve 0.085 random memory operations/sec which is the closest performance that is higher than LEAP cache's performance. The LEAP cache achieves 0.082 random memory operations/sec. In both of these cases, we used DRAM memory model described in Section 6.2.2. The generated hardware by Gorilla++ has 26% higher area than the equivalent LEAP cache. This is due to the fact that the non-blocking

Table 6.12: Comparing the area of Gorilla++-generated designs with manually-crafted baseline designs. GR: Gorilla++ refinement algorithm, SA: simulated annealing. Gorilla++ area and frequency numbers are reported relative to the baseline area and frequencies. The area and frequency numbers are from place and route report.

		Maximum frequency			Area		
Application	Performance	baseline (Mhz)	GR (relative)	SA (relative)	baseline (LUTs)	GR (relative)	SA (relative)
Non-blocking cache	0.082 million random reads / sec	160	0.75	0.75	2,010	1.26	1.26
IPv4/IPv6 router	100 million 64-bytes packets / sec	110	1.18	1.18	75,232	0.92	0.98
MPLS switch	100 million 64-bytes packets / sec	110	0.85	0.95	87,765	1.3	1.05

behavior in the LEAP cache is designed using hand-crafted MSHR (miss status holding registers) while Gorilla++ automatically generates non-blocking behavior using multi-threaded engine template.

As part of Gorilla [51] project, we implemented an IPv4/IPv6 packet processor and an MPLS switch in Verilog. We implemented the same hardware using Gorilla++ to compare the quality of the hardware generated by Gorilla++. For IPv4/IPv6 packet processor, Gorilla++ generates a hardware that has 8% lower area than the equivalent manually-crafted design. We found that the arbitration/distribution components generated by Chisel is more efficient that the arbiters we used in Verilog design. This highlights the fact that when higher-level constructs are used, the highly-efficient library components

can improve the quality of result. However, when low-level design methodologies are used, due to the higher complexity of re-using the components, the designers may tend to implement the components by themselves which may result in a lower performance. For MPLS switch, Gorilla++ generates a hardware that has 30% higher area. This is because of the inefficiency in the Gorilla++ rule-based DSE due to the use of static priorities for selecting the refinement rules (see Chapter 7).

Chapter 7

Conclusion and future work

In conclusion, the combination of streaming and shared-memory programming models is a natural way to describe many important applications in networking and big-data domains. When the programming model uses structured composition and explicit global memory, a major improvement in productivity using auto-refinements is achievable.

The Gorilla++ compiler can apply micro-architecture refinements including multi-threading, pipe-offloading, and offload-sharing. These refinements improve the area of the generated accelerators by an average of 55%.

The Gorilla++ rule-based DSE is a powerful exploration technique that iteratively eliminates the bottlenecks and under-utilized resources in the design. It uses performance counters to detect such inefficiencies. The performance counters are automatically injected and managed by the toolset. On average, the Gorilla++ rule-based DSE uses 3.4x fewer iterations compared to hill climbing while generating hardware with 9% higher utility and 6x fewer iterations compared to simulated annealing while generating hardware with 1% lower utility.

In the future, we intend to improve Gorilla++ in various directions. First, we intend to extend the Gorilla++ language constructs to support more applications. For example, supporting recursive offloads and supporting arrays of components are two extensions that can improve the generality of the

language.

Second, we intend to extend the Gorilla++ compiler refinements beyond concurrency-related refinements. For example, locality-related refinements can be used to introduce caches into micro-architectures and fine-tune the cache parameters automatically. Similarly, synchronization-related refinements can be used to introduce hierarchy into lock components and fine-tune their parameters.

Third, we intend to include an area model in the rule-based DSE to choose the appropriate rules in a dynamic fashion, rather than the current rule-selection method which is based on static priorities. In the MPLS benchmark, for example, Gorilla++ produces hardware with a lower quality than simulated annealing and hill climbing because of the lack of a dynamic rule selection model.

Appendices

Appendix A

The reference manual of Gorilla++ Engine language

⟨*program*⟩ ::= ⟨*interface-pragmas*⟩ ⟨*type-definitions*⟩ ⟨*variable-definitions*⟩
⟨*processing-steps*⟩
| ⟨*type-definitions*⟩

⟨*interface-definitions*⟩ ::= ⟨*input-definition*⟩ ⟨*output-definition*⟩
⟨*offload-definitions*⟩

⟨*input-definition*⟩ ::= '`#pragma`' '`INPUT`' '`(`' ⟨*type-ident*⟩ '`)`'

⟨*output-definition*⟩ ::= '`#pragma`' '`OUTPUT`' '`(`' ⟨*type-ident*⟩ '`)`'

⟨*offload-definitions*⟩ ::= ⟨*offload-definition*⟩ ⟨*offload-definitions*⟩
| ⟨*offload-definition*⟩

⟨*offload-definition*⟩ ::= '`#pragma` `OFFLOAD`' '`(`' ⟨*type-identifier*⟩ '`,`'
⟨*offload-ident*⟩ '`)`'

⟨*variable-definitions*⟩ ::= ⟨*variable-definition*⟩ ⟨*variable-definitions*⟩
| ⟨*variable-definitions*⟩

⟨*variable-definition*⟩ ::= ⟨*type-ident*⟩ ⟨*variable-list*⟩

⟨*varialbe-list*⟩ ::= ⟨*variable-id*⟩ ⟨*variable-list*⟩
| ⟨*variable-id*⟩

⟨*processing-steps*⟩ ::= ⟨*processing-step*⟩

| ⟨*processing-step*⟩ ⟨*processing-steps*⟩

⟨*statement-list*⟩ ::= ⟨*statement*⟩ ⟨*statement-list*⟩

| ⟨*statement*⟩

⟨*processing-step*⟩ ::= ⟨*processing-step-name*⟩ '`( )`' '`{`' ⟨*variable-definitions*⟩
⟨*statement-list*⟩ ⟨*offload-statement-list*⟩ ⟨*statement-list*⟩ '`}`'

⟨*statement*⟩ ::= ⟨*ident*⟩ '`=`' ⟨*expr*⟩ '`;`'

| ⟨*if-else-statement*⟩

| ⟨*if-statement*⟩

| '`{`' ⟨*statement-list*⟩ '`}`'

| ⟨*emit-finish-statement*⟩

⟨*if-statement*⟩ ::= '`if`' '`(`' ⟨*expr*⟩ '`)`' '`{`' ⟨*statement-list*⟩ '`}`'

⟨*if-else-statement*⟩ ::= if-statement '`else` `{`' ⟨*statement-list*⟩ '`}`'

⟨*offload-statement-list*⟩ ::= ⟨*offload-statement*⟩ ⟨*offload-statement-list*⟩

| ⟨*offload-statement*⟩

⟨*expr*⟩ ::= ⟨*ident*⟩ ⟨*binary-operator*⟩ ⟨*expr*⟩

| ⟨*ident*⟩

| ⟨*cast-prefix*⟩ ⟨*expr*⟩

| ⟨*ident*⟩ ⟨*field-expression*⟩

⟨*cast-prefix*⟩ ::= '`(`' ⟨*type-identifier*⟩ ')'

⟨*field-expression*⟩ ::= '`.`' ⟨*ident*⟩

| '`.`' ⟨*ident*⟩ ⟨*field-expression*⟩

⟨*offload-statement*⟩ ::= ⟨*ident*⟩ '`=` `<offload-ident>` `(`' ⟨*expr*⟩ '`)`' '`;`'

⟨*emit-finish-statement*⟩ ::= ‘`finish`’ ‘`(`’ ‘`)`’ ‘`;`’

| ‘`emit`’ ‘`(`’ (⟨*processing-step-name*⟩)? ‘`)`’

| ‘`finishNoEmit`’ ‘`(`’ ‘`)`’‘`;`’

⟨*type-definitions*⟩ ::= ⟨*type-definition*⟩ ⟨*type-definitions*⟩

| ⟨*type-definition*⟩

⟨*type-definition*⟩ ::= ‘`typedef`’ ⟨*type-id-new*⟩ ⟨*type-id-old*⟩

| ‘`typedef`’ ‘`struct`’ ‘`{`’ ⟨*variable-definitions*⟩ ‘`}`’ ⟨*bundle-type-id*⟩ ‘`;`’

Bibliography

[1] "Altera OpenCL," http://www.altera.com/products/software/opencl/opencl-index.html.

[2] Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: Data structures for parallel computing," *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 4, pp. 598–632, Oct. 1989.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *ACM SIGMETRICS/PERFORMANCE 2012*, ser. SIGMETRICS '12. ACM, 2012.

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: Constructing hardware in a scala embedded language," *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pp. 1212 –1221, june 2012.

[5] N. Binkert, A. Saidi, and S. Reinhardt, "Integrated network interfaces for high-bandwidth tcp/ip," *SIGARCH Comput. Archit. News*, Oct. 2006.

[6] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in haskell," *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '98, pp. 174–184. New York, NY, USA: ACM, 1998.

[7] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in haskell," *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '98, pp. 174–184. New York, NY, USA: ACM, 1998.

[8] "Bluespec webpage," http://www.bluespec.com.

[9] "C-to-Verilog synthesis tool," http://c-to-verilog.com/.

[10] A. Canis, S. Brown, and J. Anderson, "Modulo sdc scheduling with recurrence minimization in high-level synthesis," *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pp. 1–8, Sept 2014.

[11] A. Canis, J. Chois, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," *FPGA 2012*, ser. FPGA '11. New York, NY, USA: ACM, 2011.

[12] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 20, no. 9, pp. 1059–1076, Sep 2001.

[13] "CatapultC," http://calypto.com/en/products/catapult/overview/.

[14] L.-F. Chao, A. LaPaugh, and E.-M. Sha, "Rotation scheduling: a loop pipelining algorithm," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 16, no. 3, pp. 229–239, Mar 1997.

[15] G. Chen, "A short historical survey of functional hardware languages," *ISRN Electronics*, vol. 2012, April 2012.

[16] E. S. Chung, J. D. Davis, and J. Lee, "Linqits: Big data on little clients," *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, pp. 261–272. New York, NY, USA: ACM, 2013.

[17] E. Chung, P. Milder, J. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?" *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pp. 225 –236, dec. 2010.

[18] "Cisco carrier routing system wiki page," http://en.wikipedia.org/wiki/Carrier-Routing-System.

[19] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, 2011.

[20] J. Cong, M. Huang, B. Liu, P. Zhang, and Y. Zou, "Combining module selection and replication for throughput-driven streaming programs," *DATE*, pp. 1018–1023, 2012.

[21] B. Cook, J. Launchbury, and J. Matthews, "Specifying superscalar microprocessors in hawk," *In Formal Techniques for Hardware and Hardware-like Systems. Marstrand*, 1998.

[22] J. T. O. donnell, "Overview of hydra: A concurrent language for synchronous digital circuit design," *In Proceedings of the 16th International*

Parallel and Distributed Processing Symposium. IEEE Computer, pp. 249–264. Society Press, 2002.

[23] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, , and D. M. Tullsen, "Simultaneous Multithreading: A Foundation for Next-generation Processors," *IEEE Micro*, pp. 12–18, Sep./Oct. 1997.

[24] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," *SIGPLAN Not.*, vol. 47, no. 4, pp. 37–48, Mar. 2012.

[25] D. D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," *IEEE Des. Test*, vol. 11, no. 4, pp. 44–54, Oct. 1994.

[26] E. F. Girczyc, "Loop winding–a data flow approach to functional pipelining," *IEEE International Symposium on Circuits and Systems*, 1987.

[27] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented fpga computing in the streams-c high level language," *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pp. 49 –56, 2000.

[28] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor, "The greendroid mobile application processor: An architecture for silicon's dark future," *Micro, IEEE*, vol. 31, no. 2, march-april 2011.

[29] D. Greaves and S. Singh, "Designing application specific circuits with concurrent C-sharp programs," *Formal Methods and Models for Codesign*

(MEMOCODE), 2010 8th IEEE/ACM International Conference on, pp. 21–30, July 2010.

[30] A. Hagiescu, W.-F. Wong, D. Bacon, and R. Rabbah, "A computing origami: Folding streams in fpgas," *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pp. 282–287, July 2009.

[31] N. Halbwachs, "Synchronous programming of reactive systems," *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Hu and M. Vardi, Eds. Springer Berlin Heidelberg, 1998, vol. 1427, pp. 1–16.

[32] R. J. Halstead and W. Najjar, "Compiled multithreaded data paths on fpgas for dynamic workloads," *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '13, pp. 3:1–3:10. Piscataway, NJ, USA: IEEE Press, 2013.

[33] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," *ISCA 2010*, ser. ISCA '10, 2010.

[34] "Handle-C Wiki Page," http://en.wikipedia.org/wiki/Handel-C.

[35] "Hill climbing algorithm," http://en.wikipedia.org/wiki/Hillclimbing.

[36] J. C. Hoe and Arvind, "Synthesis of operation-centric hardware descriptions," *Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '00, pp. 511–519. Piscataway, NJ, USA: IEEE Press, 2000.

[37] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, "Optimus: efficient realization of streaming applications on fpgas," *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 41–50, 2008.

[38] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah, "Liquid metal: object-oriented programming across the hardware/software boundary," *Liquid Metal: object-oriented programming across the hardware/software boundary*, pp. 76–103. Springer-Verlag, 2008.

[39] "Demystifying Big Data," http://www-01.ibm.com/software/data/demystifying-big-data/.

[40] "Impulse-C," http://www.impulseaccelerated.com/.

[41] "Packet processing on Intel architecture," http://www.intel.com/p/en_US/embedded/hwsw/technology/packet-processing.

[42] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07, pp. 59–72. New York, NY, USA: ACM, 2007.

[43] R. Jagannathan and A. Faustini, "Glu: a system for scalable and resilient large-grain parallel processing," *System Sciences, 1991. Proceedings of the Twenty-Fourth Annual Hawaii International Conference on*, vol. i, pp. 36–48 vol.1, Jan 1991.

[44] R. Jain, A. Mujumdar, A. Sharma, and H. Wang, "Empirical evaluation of some high-level synthesis scheduling heuristics," *Proceedings of the*

28th ACM/IEEE Design Automation Conference, ser. DAC '91, pp. 686–689. New York, NY, USA: ACM, 1991.

[45] G. Jones and M. Sheeran, "Deriving bit-serial circuits in ruby," 1991.

[46] S.-W. Jun, M. Liu, K. E. Fleming, and Arvind, "Scalable multi-access flash store for big data analytics," *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14, pp. 55–64. New York, NY, USA: ACM, 2014.

[47] G. Kahn, "The semantics of simple language for parallel programming," *IFIP Congress*, pp. 471–475, 1974.

[48] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *SCIENCE*, vol. 220, no. 4598, pp. 671–680, 1983.

[49] D. Ku and G. DeMicheli, "Hardwarec – a language for hardware design (version 2.0)," Stanford, CA, USA, Tech. Rep., 1990.

[50] m. lavasani, H. Angepat, and D. Chiou, "An fpga-based in-line accelerator for memcached," pp. 1–1, 2013.

[51] M. Lavasani, L. Dennison, and D. Chiou, "Compiling High Throughput Network Processors," *20th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2012.

[52] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, sept. 1987.

[53] J.-H. Lee, Y.-C. Hsu, and Y.-L. Lin, "A new integer linear programming formulation for the scheduling problem in data path synthesis,"

Computer-Aided Design, 1989. ICCAD-89. Digest of Technical Papers., 1989 IEEE International Conference on, pp. 20–23, Nov 1989.

[54] S. Li, Y. Liu, X. Hu, X. He, Y. Zhang, P. Zhang, and H. Yang, "Optimal partition with block-level parallelization in c-to-rtl synthesis for streaming applications," *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pp. 225–230, Jan 2013.

[55] Y. Li and M. Leeser, "Hml: an innovative hardware description language and its translation to vhdl," *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scal*, pp. 691–696, Aug 1995.

[56] M. N. Lis, "Superscalar processors via automatic microarchitecture transformations," PhD thesis, Massachusetts Institute of Technology, 2000.

[57] F. Liu, S. Ghosh, N. P. Johnson, and D. I. August, "Cgpa: Coarse-grained pipelined accelerators," *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14, pp. 78:1–78:6. New York, NY, USA: ACM, 2014.

[58] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13, pp. 50:1–50:7. New York, NY, USA: ACM, 2013.

[59] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi,

"Scale-out processors," *SIGARCH Comput. Archit. News*, vol. 40, no. 3, pp. 500–511, Jun. 2012.

[60] "Memcached Wiki Page," http://en.wikipedia.org/wiki/Memcached.

[61] A. Mycroft and R. Sharp, "A statically allocated parallel functional language," *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, U. Montanari, J. Rolim, and E. Welzl, Eds. Springer Berlin Heidelberg, 2000, vol. 1853, pp. 37–48.

[62] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13, pp. 385–398. Berkeley, CA, USA: USENIX Association, 2013.

[63] E. Nurvitadhi, J. C. Hoe, S.-L. L. Lu, and T. Kam, "Automatic multithreaded pipeline synthesis from transactional datapath specifications," *Proceedings of the 47th Design Automation Conference*, ser. DAC '10, pp. 314–319. New York, NY, USA: ACM, 2010.

[64] M. Odersky and al., "An overview of the scala programming language," EPFL Lausanne, Switzerland, Tech. Rep. IC/2004/64, 2004.

[65] "Open onload project," http://www.openonload.org/.

[66] "Open System Interconnection Model," http://en.wikipedia.org/wiki/OSI_-model.

[67] P. R. Panda, "Systemc: A modeling platform supporting multiple design abstractions," *Proceedings of the 14th International Symposium on Systems Synthesis*, ser. ISSS '01, pp. 75–80. New York, NY, USA: ACM, 2001.

[68] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis of asics," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 8, no. 6, pp. 661–679, Jun 1989.

[69] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," *SIGPLAN Not.*, vol. 46, no. 6, pp. 12–25, Jun. 2011.

[70] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pp. 13–24, June 2014.

[71] "300Mhz Two Word Burst QDRII datasheet," http://www.cypress.com/?docID=21484.

[72] "RIPE Route Information Service," http://www.ripe.net/projects/ris/rawdata.html.

[73] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," *SIGARCH Comput. Archit. News*, vol. 28,

no. 2, pp. 128–138, May 2000.

[74] F. Sanchez and J. Cortadella, "Time-constrained loop pipelining," *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*, pp. 592–596, Nov 1995.

[75] B. Schafer, T. Takenaka, and K. Wakabayashi, "Adaptive simulated annealer for high level synthesis design space exploration," *VLSI Design, Automation and Test, 2009. VLSI-DAT '09. International Symposium on*, pp. 106–109, April 2009.

[76] B. C. Schafer and K. Wakabayashi, "Divide and conquer high-level synthesis design space exploration," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, no. 3, pp. 29:1–29:19, Jul. 2012.

[77] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. Rau, D. Cronquist, and M. Sivaraman, "Pico-npa: High-level synthesis of nonprogrammable hardware accelerators," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 31, no. 2, pp. 127–142, 2002.

[78] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. Stevenson, S. Richardson, M. Horowitz, B. Lee, A. Solomatnikov, and A. Firoozshahian, "Rethinking digital design: Why design must change," *Micro, IEEE*, vol. 30, no. 6, pp. 9 –24, nov.-dec. 2010.

[79] O. Shacham, S. Galal, S. Sankaranarayanan, M. Wachs, J. Brunhaver, A. Vassiliev, M. Horowitz, A. Danowitz, W. Qadeer, and S. Richardson, "Avoiding game over: Bringing design to the next level," *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pp. 623–629, 2012.

[80] R. Sharp and A. Mycroft, "A higher-level language for hardware synthesis," *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science, T. Margaria and T. Melham, Eds. Springer Berlin Heidelberg, 2001, vol. 2144, pp. 228–243.

[81] M. Sheeran, "ufp a algebraic vlsi design language," OUCL, Tech. Rep. PRG39, November 1983.

[82] "Simulated annealing Wiki Page," http://en.wikipedia.org/wiki/SimulatedAnnealing.

[83] S. Singh, "Designing reconfigurable systems in lava," *VLSI Design, 2004. Proceedings. 17th International Conference on*, pp. 299–306, 2004.

[84] M. Tan, B. Liu, S. Dai, and Z. Zhang, "Multithreaded pipeline synthesis for data-parallel kernels," *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '14, pp. 718–725. Piscataway, NJ, USA: IEEE Press, 2014.

[85] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," *Proceedings of the 11th International Conference on Compiler Construction*, pp. 179–196. London, UK: Springer-Verlag, 2002.

[86] "Thunk subroutines in computer programming," http://en.wikipedia.org/wiki/Thunk.

[87] "Turing machine Wiki Page," http://en.wikipedia.org/wiki/Turing-machine.

[88] "Xilinx Vivado Tool suite."

[89] H. J. Yang, K. Fleming, M. Adler, and J. Emer, "Leap shared memories: Automating the construction of fpga coherent memories," *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pp. 117–124, May 2014.

[90] Z. Zhang and B. Liu, "Sdc-based modulo scheduling for pipeline synthesis," *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '13, pp. 211–218. Piscataway, NJ, USA: IEEE Press, 2013.

Vita

Maysam Lavasani was born in Tehran, Iran in 1976, son of Forough Rezai and Hossein Lavasani. He earned B.Sc. degree in Computer Engineering from Sharif University of Technology, Tehran, Iran, in 1998. He received his M.Sc. degree in Electrical and Computer Engineering from University of Tehran, Tehran, Iran, in 2001. Maysam has worked in the area of information and communication technology in Iran from 1998 to 2007. He has been a graduate student at the University of Texas at Austin since 2008.

Permanent address: maysamlavasani@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.